Titre: Title:	Étude des cycles des codes correcteurs d'erreurs LDPC et convolutionnels doublement orthogonaux
Auteur: Author:	Benjamin Baqué
Date:	2008
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Baqué, B. (2008). Étude des cycles des codes correcteurs d'erreurs LDPC et convolutionnels doublement orthogonaux [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/8316/

Université de Montréal

ÉTUDE DES CYCLES DES CODES CORRECTEURS D'ERREURS LDPC ET CONVOLUTIONNELS DOUBLEMENT ORTHOGONAUX

par
BENJAMIN BAQUÉ

DEPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTREAL

MÉMOIRE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE MAITRISE ÈS SCIENCES APPLIQUÉES (M.SC.A)
(GÉNIE ÉLECTRIQUE)
AOÛT 2008

Your file Votre référence
ISBN: 978-0-494-46032-0
Our file Notre référence
ISBN: 978-0-494-46032-0

ÉCOLE POLYTECHNIQUE DE MONTREAL
UNIVERSITE DE MONTREAL

Ce mémoire intitulé :

ÉTUDE DES CYCLES DES CODES CORRECTEURS D'ERREURS LDPC ET CONVOLUTIONNELS DOUBLEMENT ORTHOGONAUX

Présentée par :
BENJAMIN BAQUÉ

En vue de l'obtention du diplôme de: Maitrise ès sciences appliquées
a été évalué par un jury composé des personnes suivantes :

M. Christian CARDINAL, Ph.D, président-rapporteur
M. David HACCOUN, Ph.D, directeur de recherche
M. Jean François FRIGON, Ph.D, membre du jury

À mon pére

REMERCIEMENTS

Je tiens à remercier par ces quelques lignes les personnes sans qui ce travail de recherche n'aurait pas pu être mené à bien.

Mes premiers remerciements vont à mon père qui n'a jamais cessé pendant toutes ces années de me pousser et de me convaincre que j'étais capable de faire plus que je ne le pensais. Il a toujours était là pour moi et je n'oublierais pas son soutien.

Je souhaite également remercier mon directeur de recherche, le Dr. David Haccoun, dont le soutien précieux m'a aidé tout au long de mon travail. Je tiens également à le remercier pour l'aide financière qu'il m'a accordée pendant la durée de ma recherche.

Merci aussi à mes collègues de laboratoire, pour leur bonne humeur quotidienne et toutes ces discussions enrichissantes. Je pense particulièrement à Samuel, Nathalie, Laurent, Zouheir et Éric.

Finalement, je tiens à remercier ma famille ainsi que mes amis pour leurs soutiens.

RÉSUMÉ

Le travail de recherche présenté dans ce mémoire concerne l'étude des cycles des graphes de décodage des codes LDPC et convolutionnels doublement orthogonaux.

Les cycles de petite longueur sont connus pour leur rôle important dans la dégradation des performances d'erreur. Lors du décodage itératif d'un code, les algorithmes de décodage ont besoin d'avoir une certaine indépendance entre les symboles reçus du canal, pour parvenir à corriger ces derniers de la façon la plus optimale possible. La présence de petits cycles, c'est-à-dire des cycles de longueur quatre, entraîne une corrélation des observables, dès la $2^{éme}$ itération. Il est donc utile d'augmenter la longueur du cycle le plus petit pour parvenir à une amélioration importante du message décodé lors de la deuxième itération. Notre but a donc été d'étudier théoriquement le comportement de codes correcteurs d'erreur contenant des cycles et d'en observer les performances d'erreur.

La première partie de notre travail consiste à décrire de manière théorique le comportement et le fonctionnement de décodeurs pour les codes convolutionnels doublement orthogonaux et LDPC. Le lien avec les cycles est fait et introduit les deux parties suivantes du mémoire.

La deuxième partie de notre travail consiste à analyser les cycles des codes LDPC avec un décodage de type Somme-Produit. Une nouvelle méthode de construction de matrices de parité est suggérée et les codes issus de cet algorithme sont comparés avec les meilleurs codes connus. De plus, une méthode d'analyse des performances de matrice de parité LDPC sans simulation est proposée.

La troisième partie de notre travail à quand à elle permis d'analyser les performances d'erreur du décodeur itératif des codes convolutionnels doublement orthogonaux en fonction des cycles du code. Plusieurs tendances sont observées et des irrégularités de construction chez les codes simplifiés sont mises en évidence.

Au cours de ces étapes, des algorithmes basés sur le comptage de cycles ont été utilisés pour l'analyse fine des matrices de parité ou génératrice. Un algorithme très performant pour le comptage des cycles de longueur minimale est aussi proposé. Notre travail d'analyse des cycles semble avoir été fructueux du fait qu'il a permis de mettre au point une nouvelle méthode de construction de matrices de parité LDPC, et un algorithme d'évaluation des performances d'erreur d'un code sans simulation.

ABSTRACT

The research presented in this master's thesis deals with the study of the cycles present in the decoding graphs of both LDPC and convolutional self doubly orthogonal codes.

The small cycles are known for their important role in the degradation of the bit error rates. During the iterative decoding of a message, the decoding algorithm requires a certain independence between the received symbols in order to correct the messages in the best way possible. The presence of small cycles, i.e. cycles of length 4, have the effect of correlating the symbols received from the channel, during the second iteration. It is therefore useful to increase the length of the smallest cycles in order to significantly improve the decoding message at the second iteration. The goal was therefore to theoretically study the behavior of error correcting codes containing cycles and to observe the ensuing bit error rates.

The first section of this work consists of theoretically describing the behavior and function of the decoders for both LDPC and convolutional self doubly orthogonal codes. The link with the cycles is highlighted and introduces the two following sections of the work.

The second part of this work consists of analyzing the cycles of LDPC codes with the Sum-Product algorithm. A new method of construction of parity check matrices is suggested and the codes resulting from this algorithm are compared with the best codes known at this time. Furthermore, a method to estimate the error performance of LDPC codes without simulation is proposed.

The third part of this work analyzes the performance of the iterative decoder of the convolutional self doubly orthogonal codes according to the cycles. Some regularities and irregularities are observed in the construction of the simplified codes.

Algorithms based on cycles counting were used to precisely analyze the parity check and generating matrix of the code. A high performance algorithm used for cycles counting is proposed. The cycle analysis appears to be fruitful due to the fact that it allowed a new construction method of the LDPC parity check matrix, without foregoing the evaluation of the error performance of a code without using computer simulation.

TABLE DES MATIERES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ANNEXES

LISTE DES SIGLES ET DES SYMBOLES

Sigles

BER	Bit Error Rate
BPSK	Binary Phase Shift Keying
BSC	Binary symmetric channel
CSOC	Convolutional Self-Orthogonal Code
CSO2C	Convolutional Self-Doubly Orthogonal Code
CSO2C-WS	Convolutional Self-Doubly Orthogonal Code-Wide Sence
LRV	Logarithme du Rapport de Vraisemblance
MAP	Maximum A *posteriori* Probability
PEG	Progressive Edge Growth
PER	Progressive Edge Removal
Perf	Performance d'erreur
S-CSO2C-WS	Simplified Convolutional Self-Doubly Orthogonal Code
SNR	Signal-to-Noise Ratio

Symboles

E_b	Energie du symbole binaire u_i
$\mathbf{v} = (v_1\ v_2\ \ldots\ v_n)$	Vecteur de dimension n et ses composantes
R	Taux de codage
J	Nombre de connexions d'un codes convolutionnel systématique de taux de codage R = ½.
$\boldsymbol{H}$	Matrice de parité
$L(B_i)$	Logarithme du rapport de vraisemblance de la variable aléatoire binaire B_i
$Q(x)$	$\frac{1}{\sqrt{2\pi}}\int_x^{\infty} \exp\left(-\frac{z^2}{2}\right) dz, \qquad x \geq 0$
$\oplus$	Somme modulo-2
$\diamondsuit$	Opérateur add-min
gnc	Cycle minimum moyen sur les nœuds de parité
gnv	Cycle minimum moyen sur les nœuds de variable
ge	Cycle minimum moyen sur liens
cmm	Cycle Minimum Moyen
$\boldsymbol{G}$	Matrice génératrice
G	Longueur du plus petit cycle (*Girth*)
$N1, N2$	Identifie les deux nœuds d'un lien
n, k	Dimension des matrices de parité **H**
n, r	Nombre de nœuds de variable et de parité $(r = k)$
O	Nombre de 1's par colonne dans **H**

CHAPITRE 1

INTRODUCTION

1.1 Motivations

Depuis plusieurs années, nous assistons à une expansion très importante des télécommunications dans la vie courante de tout en chacun. De grandes avancés technologiques dans l'électronique ont permis de démocratiser les moyens de communications modernes comme l'Internet et les téléphones portables. Même les télécommunications par satellite, avec des applications comme la télévision, la radio et la téléphonie ont su s'adapter aux besoins des consommateurs.

Cependant, tous ces modes de communications sont contraints à des limitations de largeur de bande. Les fournisseurs de services, qui doivent innover en permanences pour rester attractifs, ont besoin de faire passer de plus en plus d'information dans les canaux dont la largeur de bande est limitée, avec une probabilité d'erreur qui satisfait une valeur cible donnée.

La théorie de l'information, établie en 1948 par Shannon [1], à l'origine des techniques de codage de canal, est la solution pour pouvoir transmettre plus d'information en limitant les erreurs, dans ces canaux bruités à largeur de bande limitée.

À cause de la présence d'erreurs de transmission à la sortie des canaux et du besoin d'en réduire leur nombre de manière significative pour un fonctionnement normal des appareils de télécommunication, l'utilisation du codage de canal pour corriger ces erreurs s'impose.

Cette protection contre les erreurs permet d'économiser de l'énergie lors de la transmission, mais consomme plus de largeurs de bande, en envoyant des symboles supplémentaires utilisés à la réception pour corriger les erreurs de transmission.

Des techniques de codage de canal ont été développées pour répondre à cette demande [1]. Les plus performantes sont, celles qui permettent d'arriver pour une probabilité d'erreur donnée, à réduire au maximum les symboles supplémentaires transmis, sur un canal donné, pour la correction des erreurs.

Les techniques de codage de canal les plus puissantes à ce jour sont les codes Turbo et les codes LDPC (Low Density Parity Check) [2]. Ce sont des codes à décodage itératifs sur graphe.

Pour avoir de très bonnes performances, il est nécessaire d'optimiser le graphe de décodage. Cependant par la présence de cycles dans ces graphes, l'optimalité est perdu et les performances d'erreur se dégradent. Même s'il n'est plus nécessaire de démontrer que la présence de cycles dégrade les performances, leur influence précise est mal connue.

Les objectifs de ce mémoire ont donc été d'étudier l'influence de ces cycles sur les performances d'erreur et d'en développer des outils d'analyse. Une méthode d'optimisation de graphe de décodage existant par l'étude des cycles, pour les codes LDPC, a même été développée. Cette méthode permet d'améliorer de manière significative les performances d'erreur des codes LDPC de [3] à haut SNR.

1.2 Contributions

Les contributions apportées par ce travail de recherche sont les suivantes.

1. Élaboration de définitions et de propriétés associées au comptage de cycles par l'exploration du graphe de décodage.
2. Création de méthodes et conception de programmes d'analyse des cycles pour les codes LDPC et les codes CSO2C-WS [4].
3. Conception de programmes permettant le comptage des cycles de longueur minimum ainsi que ceux de longueur 4, 6 et 8.

4. Élaboration d'un algorithme d'amélioration des matrices de parités des codes LDPC contenant des cycles de longueur quatre.

Toutes les simulations ont été effectuées à l'aide d'ordinateurs munis de processeurs pentium® IV d'Intel® cadencés de 2.6 Ghz à 3.2 Ghz, sur des plateformes mono et biprocesseur, et possédant de 1Gb à 2Gb de RAM sous les environnements Windows XP© et LINUX 2.6.21. Les langages de programmation Matlab©, C++, Java© et Visual Basic ont servi à la conception des algorithmes de simulation, d'analyse, de recherche et de calculs.

1.3 Organisation du mémoire

Ce mémoire comporte sept chapitres. À la suite du présent chapitre, le document se subdivise de la façon suivante.

- Le chapitre 2 présente les notions élémentaires associées aux codes convolutionnels doublement orthogonaux et aux codes en bloc. La structure de décodage à seuil des codes convolutionnels doublement orthogonaux est présentée. Par la suite, nous introduisons brièvement la représentation des codes sous forme de graphe biparti.
- Le chapitre 3 présente les notions élémentaires associées aux codes à faible densité de parité (LDPC). Plusieurs méthodes de construction y sont présentées et l'algorithme Somme-Produit utilisé par le décodeur est introduit.
- Le chapitre 4 définit les nombreux algorithmes développés pour le comptage et l'étude des cycles. Certains problèmes liés aux grandes tailles des graphes bipartis y sont évoqués. Les définitions et les démonstrations nécessaires à valider les méthodes de comptage sont présentées. Enfin, la notion de cycle minimum moyen (CMM) est définie.

- Le chapitre 5 résume les différentes études que nous avons menées sur les codes LDPC. Plusieurs sujets sont traités dans cette partie, en particulier notre méthode d'optimisation de matrice de parité (PER), l'étude du cycle minimum moyen sur les performances d'erreur, ou encore l'étude du nombre de petits cycles. Une comparaison est faite les codes PER et les meilleurs codes connus.
- Le chapitre 6 résume les différentes études que nous avons menées sur les codes convolutionnels doublement orthogonaux au sens large CSO2C-WS et S-CSO2C-WS. Une analyse des cycles des codes CSO2C récursifs est faite. L'évolution du nombre de cycles sur les CSO2C-WS et S-CSO2C-WS est réalisée et l'irrégularité de la distribution de leurs cycles dans le graphe de décodage est étudiée.
- Finalement, le chapitre 7 résume l'ensemble des travaux effectués et propose certaines idées à étudier dans le futur.

CHAPITRE 2

LES CODES CORRECTEURS D'ERREURS

2.1 Introduction

Les systèmes de communication numérique sont utilisés pour transmettre des messages d'une source vers une destination. À cause du bruit qui s'ajoute au signal, l'ensemble des systèmes de communication moderne supportent des techniques de codage protégeant l'intégrité de l'information transmise dans le canal. Les codes convolutionnels et les codes en blocs sont les deux principales techniques utilisées pour la protection de l'information.

Les codes correcteurs d'erreur permettent d'accroître la fiabilité des systèmes de communication numérique. Cette tâche est effectuée grâce à une redondance de l'information transmise. Le codage consiste donc à envoyer l'information utile avec des bits supplémentaires connus sous le nom de bit de parité, calculés selon des règles bien précises connues de l'émetteur et du récepteur.

Comme nous l'avons mentionné plus haut, il existe deux grandes familles de codes : les codes en bloc et les codes convolutionnels.

Le premier encode des blocs d'information de manière indépendante par l'ajout de plusieurs symboles de parité à la séquence d'information. Cela implique la nécessité d'avoir reçu l'intégralité du bloc pour procéder à son décodage.

Le deuxième code l'information de façon continue par l'utilisation de registres à décalage. Pour cette raison, le décodage est possible dès qu'une quantité suffisante de bits est arrivée au récepteur. Il s'agit de la technique de codage la plus efficace dans les communications temps réels. Ce second chapitre décrit les caractéristiques de base des différents systèmes de codage utilisés dans ce mémoire, tels que les codes convolutionnels doublement orthogonaux [5] et les codes à faible densité de parité (LDPC) [2].

2.2 Système de communication numérique

Nous vous présentons dans cette section, le système de communication numérique utilisé dans ce mémoire. Ce modèle est illustré par le schéma de la Figure 2.1.

Une source émet des séquences de symboles binaires $\boldsymbol{u} = (u_0, u_1, \ldots)$ chacun d'énergie E_b vers un codeur de taux de codage R. Ce codeur ajoute un ou plusieurs bits de redondance pour chaque bit d'information émis par la source et forme ainsi la séquence binaire codée $\boldsymbol{v} = (v_0, v_1, \ldots)$. Cette séquence est ensuite envoyée dans le canal avec par exemple une modulation BPSK qui n'est rien d'autre qu'une modulation antipodale, faisant correspondre les bits 0 à une tension négative et les bits 1 à la même tension, mais positive. La tension $S_i(t)$ à la sortie du modulateur BPSK est régie par l'équation suivante :

$$S_i(t) = x_i . f(t) = (2v_i - 1).f(t) \quad \text{avec } f(t) = 1, \forall t \in [0, T] \tag{2.1}$$

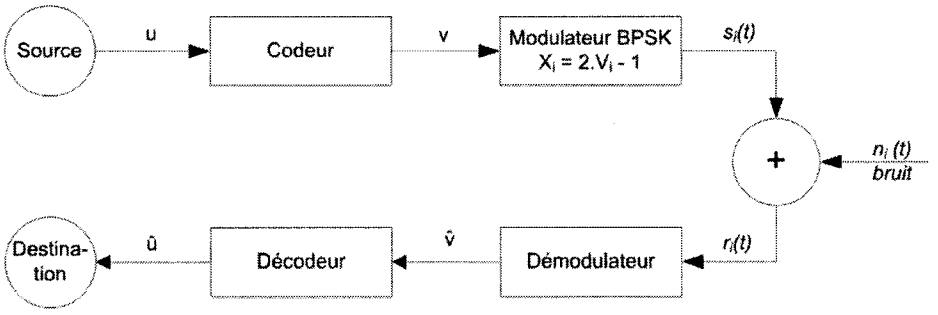


Figure 2.1: Modèle d'une communication numérique

Le canal considéré est binaire symétrique (en anglais Binary Symmetric Channel BSC) à bruit banc additif Gaussien. Il est dit sans mémoire car les symboles transmis dans le canal sont statistiquement indépendants. Le bruit $n_i(t)$ est un processus aléatoire Gaussien de moyenne nulle et de densité spectrale unilatérale N_0Watts/Hz. Le rapport

signal à bruit est défini par E_b/N_0. Ce rapport est utilisé dans l'évaluation des performances des systèmes de communications numériques.

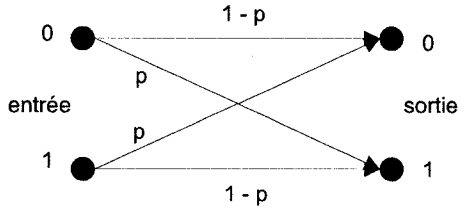


Figure 2.2: Canal BSC discret sans mémoire.

Le signal reçu $r_i(t) = s_i(t) + n_i(t)$ est démodulé par l'utilisation d'un filtre adapté. En raison des erreurs induites par le bruit du canal $n_i(t)$, la sortie du démodulateur à l'instant i est alors :

$$\tilde{v}_i = v_i \oplus e_i \tag{2.2}$$

dans le cas d'une quantification dure où e_i représente l'erreur qui affecte le symbole codé v_i. Sur la Figure 2.2, la probabilité que le récepteur reçoive un symbole à 1 sachant qu'un symbole 0 à été transmis est égale à la probabilité de transition p. Cette probabilité dépend de l'énergie du signal Es ainsi que de la variance du bruit N_0. Pour une modulation BPSK dans un canal BSC sans mémoire et à bruit blanc additif gaussien, la probabilité de transition p est donnée par [6] :

$$p = \mathbb{Q}\left(\sqrt{\frac{2.Es}{N_0}}\right) \tag{2.3}$$

Une fois la démodulation achevée, la séquence de la sortie quantifiée $\tilde{v} = (\tilde{v}_0, \tilde{v}_1, \ldots)$ est transmise au décodeur qui en utilisant les symboles redondants, parviendra à décoder avec plus de fiabilité les symboles d'information.

2.3 Codes convolutionnels

Les codes convolutionnels sont une classe de codes correcteurs d'erreurs qui génèrent un symbole de parité pour chaque symbole d'information émanant de la source. Les codeurs convolutionnels codent les bits d'information de façon continue, permettant ainsi un décodage minimisant la latence au niveau du récepteur, contrairement à leurs homologues en blocs. Le codeur est constitué de k registres à décalage composé de K cellules connectées à n additionneurs modulo-2 comme illustré à la Figure 2.3.

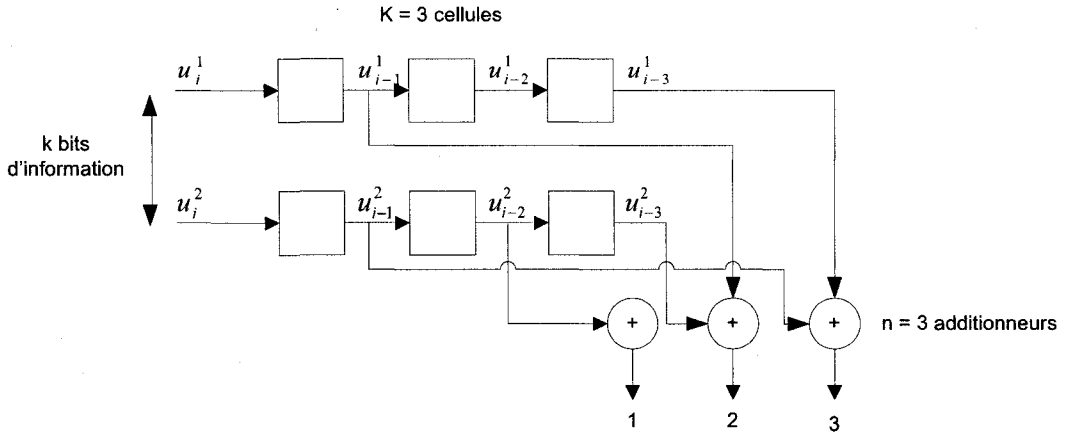


Figure 2.3: Exemple de codeur convolutionnel de taux de codage $R = \frac{k}{n} = \frac{2}{3}$

Cependant, pour ce mémoire nous nous intéresserons presque exclusivement à des codes convolutionnels systématiques avec un taux de codage $R = ½$.

Un codeur convolutionnel systématique reprend le même principe qu'un codeur conventionnel et ajoute à l'une de ses sorties la séquence d'entrée. La Figure 2.4 illustre un codeur convolutionnel systématique avec un taux de codage ½.

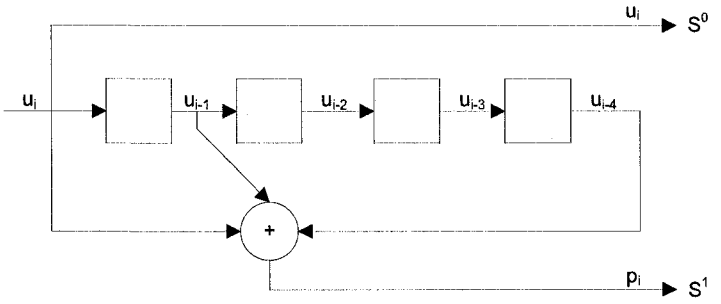


Figure 2.4: Exemple de codeur convolutionnel systématique de taux de codage ½.

Ce codeur génère deux sorties dont l'une d'entre elle reprend les bits d'entrée du codeur et l'autre la sortie de l'additionneur modulo-2. Les deux séquences de sorties $\boldsymbol{S^0}$ et $\boldsymbol{S^1}$ sont ensuite multiplexées afin de respecter l'ordonnancement présenté en (2.4).

$$\boldsymbol{S} = (u_1, p_1, u_2, p_2, u_3, p_3 \ldots) \tag{2.4}$$

Contrairement aux codes en blocs qui sont généralement définis par une matrice binaire, les codes convolutionnels peuvent être définis par un ensemble de connexions reliant les registres à décalage aux additionneurs modulo-2. Pour l'exemple de la Figure 2.4, l'ensemble $\boldsymbol{A}$, défini par $\boldsymbol{J}$ nombres entiers positifs $\boldsymbol{\alpha_k}$ définissant les connexions sur les registres est donnée par :

$$A = \{\alpha_1, \alpha_2, \alpha_3\} = \{0, 1, 4\} \tag{2.5}$$

En utilisant cet ensemble de connexion $\boldsymbol{A}$, il est possible de calculer les symboles de parité par la formule (2.6) :

$$p_i = \sum_{k=1}^{J} \mathrm{u_{i-\alpha_k}} \tag{2.6}$$

L'entier le plus grand (α_J) est relié à la mémoire m du codeur convolutionnel. Cette mémoire est directement reliée aux délais de décodage du code, ou encore, au nombre de cellules du registre à décalage. Elle s'exprime par la formule :

$$m = \alpha_J \tag{2.7}$$

2.3.1 Générateur et codes convolutionnels

Les codes convolutionnels peuvent aussi être générés par une matrice comme le sont les codes en bloc. Soit la séquence d'information $\boldsymbol{U} = (u_1, u_2, \ldots, u_j)$ où $u_i \epsilon \{0, 1\}$

Le processus de codage avec les registres à décalage s'effectue et l'on obtient la séquence codée suivante, à l'instant t :

$$\boldsymbol{X}_t^{(i)} = \sum_{j=0}^{m} \oplus u_{t-j} \,.\, g_{i,j} \quad i = 1,2 \ \textit{car il y a deux sorties au codeur} \tag{2.8}$$

où $g_{i,j}$ représente une connexion entre le i^{ème} additionneur modulo-2 et la j^{ème} cellule du registre à décalage.

En introduisant D, appelé « unité de retard » ou « délai », la relation précédente devient :

$$\boldsymbol{X}_i = \boldsymbol{G}_i(D) \,.\, \boldsymbol{U} \tag{2.9}$$

On peut alors définir le polynôme générateur $\boldsymbol{G}_i(D)$ associé au $i^{ème}$ additionneur modulo-2 de la manière suivante :

$$\boldsymbol{G}_i\ (D) = \sum_{j=0}^{m} \oplus D^j \,.\, g_{i,j} \qquad i = 1,2 \tag{2.10}$$

On définit aussi $\boldsymbol{G_i}$, les vecteurs générateurs du code:

$$\boldsymbol{G_i} = [g_{i0}\, g_{i1} \ \ldots \ g_{im}] \quad avec\, i = 1,2 \tag{2.11}$$

Comme le codeur est systématique, les bits de la sortie $i = 1$ sont identiques aux bits d'entrées du codeur. Cela revient à la relation :

$$g_{1j} = u_j \tag{2.12}$$

La composante g_{2j}, est égale à 1 si la $j^{ème}$ cellule du registre à décalage est connectée au sommateur modulo-2. Dans le cas contraire, elle vaut 0.

On regroupe ensuite l'ensemble des vecteurs $\boldsymbol{G_i}$ dans la matrice génératrice $\boldsymbol{G}$:

$$\boldsymbol{G} = [\boldsymbol{G_1}, \boldsymbol{G_2}] \tag{2.13}$$

L'équation précédente devient alors :

$$\boldsymbol{G} = \begin{bmatrix} g_{10}\, g_{20} & g_{11}\, g_{21} & \cdots & g_{1m}\, g_{2m} \\ 0\ 0 & g_{10}\, g_{20} & \ddots & \vdots \\ 0\ 0 & 0\ 0 & \ddots & g_{11}\, g_{21} \\ 0\ 0 & 0\ 0 & & g_{10}\, g_{20} \end{bmatrix}$$

Cette matrice génératrice **G** peut être utilisée pour coder les séquences d'information. Pour chaque séquence d'information $\boldsymbol{u}$, de dimension $\boldsymbol{M}$, un mot de code $\boldsymbol{v}$ lui est associé. Il est alors possible de calculer la séquence de sortie $\boldsymbol{v}$ du codeur par la relation :

$$\boldsymbol{v} = \boldsymbol{u}.\boldsymbol{G} \tag{2.14}$$

Exemple : Pour le code $A = \{0, 1, 4\}$, on obtient la matrice suivante :

$$G = \begin{bmatrix} 11 & 01 & 00 & 00 & 01 \\ 00 & 11 & 01 & 00 & 00 \\ 00 & 00 & 11 & 01 & 00 \\ 00 & 00 & 00 & 11 & 01 \\ 00 & 00 & 00 & 00 & 11 \end{bmatrix}$$

En pratique, pour obtenir la première ligne de la matrice, il suffit d'envoyer un 1 à l'entrée du codeur et de réceptionner les bits qui en sortent, formant ainsi la première ligne. Les lignes suivantes s'obtiennent ensuite par simple décalage de la précédente.

2.3.2 Codes doublement orthogonaux

Dans cette partie, nous présentons brièvement les codes doublement orthogonaux [7] qui sont utilisés dans ce mémoire pour l'étude des cycles. Cette famille de codes convolutionnels a été proposé par C. Cardinal, D. Haccoun, F. Gagnon et N. Batani en 1997 pour concurrencer les codes Turbo [8]. Les problèmes inhérents à ces derniers, comme la latence due aux entrelaceurs et la complexité de décodage, ont été améliorés. L'algorithme de décodage des codes Turbo MAP [8] est remplacé par un décodeur à seuil itératif plus simple. Afin de réduire la latence, les entrelaceurs sont supprimés. Une double orthogonalité au niveau des connexions sur les registres à décalage est mise en place afin de garantir l'indépendance des observables que garantissait le système d'entrelaceurs. Nous allons par la suite décrire le décodage à seuil utilisé pour ces codes et les conditions de double orthogonalité.

2.3.2.1 Conditions de double orthogonalité au sens strict

La double orthogonalité au niveau des connexions sur les registres du générateur est nécessaire pour garantir l'indépendance des symboles codés. Pour cela, les connexions doivent respecter trois règles afin de garantir l'indépendance des observables jusqu'à la deuxième itération de l'algorithme de décodage.

Un code convolutionnel systématique de taux R = ½ est dit doublement orthogonal au sens strict si et seulement si :

- les différences $\alpha_{v,n} - \alpha_{k,n}$ sont distinctes
- les différences doubles $(\alpha_{v,n} - \alpha_{k,n}) - (\alpha_{l,s} - \alpha_{k,s})$ sont distinctes des différences simples.
- Les différences doubles $(\alpha_{v,n} - \alpha_{k,n}) - (\alpha_{l,s} - \alpha_{k,s})$ sont distinctes entre elles.

  où $(v, n, s, k, l) \epsilon \{1, 2, \dots, J\}$ tels que $s \neq n,\ k \neq v,\ k \neq l.$

Cette définition [5] de double orthogonalité garanti l'indépendance des observables à la deuxième itération par rapport à la première.

2.3.2.2 Conditions de double orthogonalité au sens large

Un code convolutionnel systématique de taux R = ½ est doublement orthogonal au sens large si et seulement si les positions $\alpha_j, j = 1, 2..J$ satisfont les propriétés suivantes :

- les différences $\alpha_j - \alpha_k$ sont distinctes
- les différences doubles $(\alpha_j - \alpha_k) - (\alpha_m - \alpha_l)$ sont distinctes des différences simples.
- Les différences doubles $(\alpha_j - \alpha_k) - (\alpha_m - \alpha_l)$ sont distinctes entre elles.

  où $(j, k, l, m) \epsilon \{1, 2, \dots, J\}$ tels que $j \neq k,\ l \neq m,\ k \neq l,\ m \neq j$

Cette définition de double orthogonalité [5][9] exclut les répétitions inévitables et garanti l'indépendance des observables à la deuxième itération par rapport à la première.

2.3.2.3 Décodage à seuil à sorties pondérées

Le décodeur à seuil à sorties pondérées est utilisé dans le décodeur des codes doublement orthogonaux. Il introduit une estimation sur le symbole décodé $\hat{u}_i$ et calcule la fiabilité de chaque équation de parité en utilisant les probabilités A *postériori* (APP).

Il utilise un système d'équations de parité $B_{i,k}$ composé de $J+1$ équations [4][5] déduites des équations de parité orthogonale $A_{i,k}$ du décodeur à seuil à quantifications fermes dont voici l'équation :

$$A_{i,k} = e_i^u \oplus e_{i+\alpha_k}^p \oplus \sum_{l=1}^{k-1} \oplus e_{i+(\alpha_k-\alpha_l)}^u \oplus \sum_{l=k+1}^{J} \oplus \left(e_{i+(\alpha_k-\alpha_l)}^u \oplus \hat{e}_{i+(\alpha_k-\alpha_l)}^u\right) \tag{2.15}$$

Les équations $B_{i,k}$ sont alors déduites en excluant le symbole $\tilde{u}_i$ de l'équation. Le système d'équations suivant est alors obtenu :

$$\begin{aligned} B_{i,k} &= A_{i,k} \oplus \tilde{u}_i \\ &= u_i \oplus e_{i+\alpha_k}^p \oplus \sum_{l=1}^{k-1} \oplus e_{i+(\alpha_k-\alpha_l)}^u \oplus \sum_{l=k+1}^{J} \oplus \left(e_{i+(\alpha_k-\alpha_l)}^u \oplus \hat{e}_{i+(\alpha_k-\alpha_l)}^u\right) \end{aligned} \tag{2.16}$$

où $k = \{0, 1, \ldots J\}$. Avec ce système d'équations, le décodeur est capable de calculer les probabilités A *postériori*, associées au symbole décodé $\hat{u}_i$.

Le décodeur décidera $\hat{u}_i = 1$ si l'équation (2.17) est vérifiée. Sinon, il choisira $\hat{u}_i = 0$.

$$Pr\left(u_i = 1 \,|\, \{B_{i,k}\}\right) \geq Pr\left(u_i = 0 | \{B_{i,k}\}\right) \tag{2.17}$$

Ce qui revient en prenant le logarithme du rapport de vraisemblance de l'équation précédente à choisir $\hat{u}_i = 1$ si:

$$\mathrm{L}(u_i|\{B_{i,k}\}) = ln\left(\frac{Pr(u_i=1|\{B_{i,k}\})}{Pr(u_i=0|\{B_{i,k}\})}\right) \geq 0 \tag{2.18}$$

Il est possible de simplifier l'écriture de ce logarithme par l'introduction de l'opérateur add-min noté ◊. Il est utilisé pour le calcul des logarithmes de rapport de vraisemblances. Soit ξ_1 et ξ_2 deux variables aléatoires binaires, d'après [9], on peut faire l'approximation suivante:

$$L(\zeta_1 \oplus \zeta_2) \approx -signe(L(\zeta_1))signe(L(\zeta_2))\min(|L(\zeta_1)|, |L(\zeta_2|) \triangleq L(\zeta_1) \lozenge L(\zeta_2) \tag{2.19}$$

Il est maintenant possible d'obtenir l'approximation λ_i du logarithme du rapport de vraisemblance qui s'écrit d'après [10]

$$\lambda_i = y_i^u + \sum_{j=1}^{J}\left(y_{i+\gamma_j}^p \lozenge \sum_{k=1}^{j-1} \lozenge\, y_{i+\gamma_j-\gamma_k}^u \lozenge \sum_{k=j+1}^{J} \lozenge\, \lambda_{i+\gamma_j-\gamma_k}\right) \tag{2.20}$$

Ce qui peut s'écrire aussi :

$$\lambda_i = y_i^u + \sum_{j=1}^{J} \psi_{i,j} \tag{2.21}$$

avec : $$\psi_{i,j}^{(\mu)} = y_{i+\gamma_j}^p \lozenge \sum_{k=1}^{j-1} \lozenge\, \lambda_{i+\gamma_j-\gamma_k}^{(\mu-1)} \lozenge \sum_{k=j+1}^{J} \lozenge\, \lambda_{i+\gamma_j-\gamma_k}^{(\mu)}$$

Cela permet dans le cas de la Figure 2.4 d'obtenir les équations de parité suivantes :

$$\begin{cases} \psi_{i,1} = y_i^p \lozenge \lambda_{i-1} \lozenge \lambda_{i-4} \\ \psi_{i,2} = y_{i+1}^p \lozenge y_{i+1}^u \lozenge \lambda_{i-3} \\ \psi_{i,3} = y_{i+4}^p \lozenge y_{i+4}^u \lozenge y_{i+3}^u \end{cases} \tag{2.22}$$

Le schéma associé à ces équations est donné à la Figure 2.5.

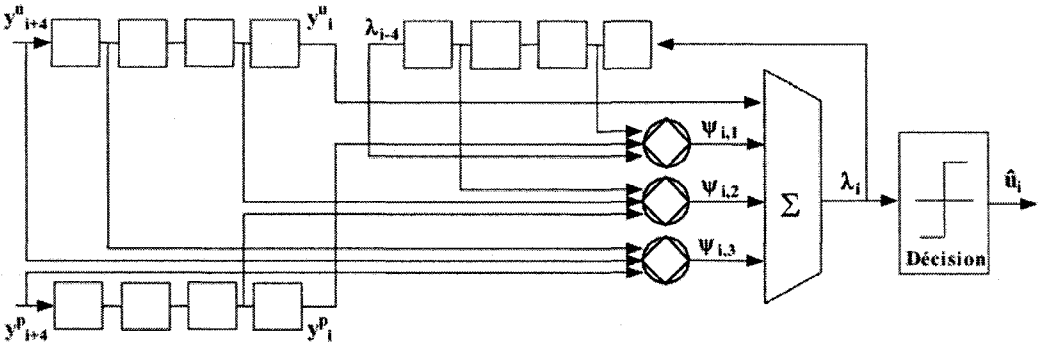


Figure 2.5: Décodeur à seuil à sortie pondéré du code $A = \{0, 1, 4\}$ de [4]

La sortie λ_i indique la fiabilité de la décision faite sur le symbole d'information $\hat{u}_i$. Cette sortie est pondérée puis utilisée par le décodeur dans le cas d'itérations multiples au décodeur.

2.3.2.4 Décodage à seuil itératif

Le décodage à seuil itératif est un algorithme générant une décision après chaque symbole reçu. Il est constitué d'une succession de décodeurs à seuil à sorties pondérées. Le schéma bloc de ce décodeur est présenté à la Figure 2.6.

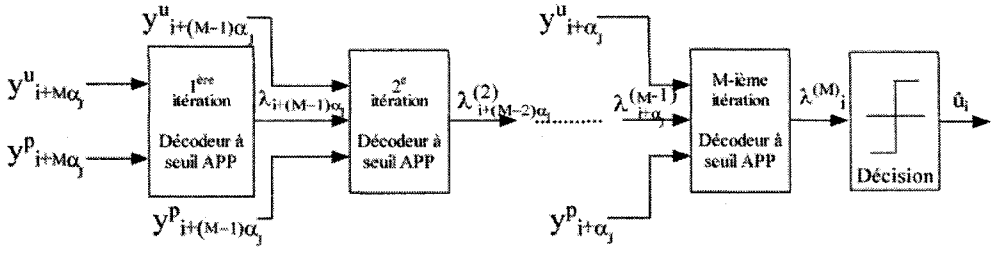


Figure 2.6: Décodeur à seuil itératif de [5]

Pour chaque itération, le décodeur calcule grâce à la sortie pondérée précédente et des symboles du canal, la sortie pondérée $\lambda_i^{(\mu)}$ associée au symbole u_i. D'après [5], la sortie pondérée à la dernière itération, notée M, peut s'écrire :

$$\lambda_i^{(M)} = y_i^u + \sum_{k=1}^{J} \left(y_{i+\alpha_k}^p \lozenge \sum_{l=1}^{k-1} \lozenge\, y_{i+\alpha_k-\alpha_l}^{(M-1)} \lozenge \sum_{l=k+1}^{J} \lozenge\, \lambda_{i+\alpha_k-\alpha_l}^{(M)} \right) \tag{2.23}$$

Cette équation garantit l'indépendance des observables jusqu'à la deuxième itération, de l'algorithme de décodage.

2.4 Codes en bloc

Les codes en blocs ont été découverts dans les années 40 par Hamming [11] et furent les premiers utilisés pour leur simplicité d'implémentation et de mise en œuvre.

Chaque mot du code en bloc (n, k) est représenté par un des 2^k vecteurs de dimension n. Il est alors possible de générer une matrice $\boldsymbol{G}_{k,n}$ qui fera correspondre chaque mot d'information à un mot de code spécifique à condition que toute combinaison linéaire de deux mots de code donne un mot de code. Le taux de codage ainsi obtenu est donné par la formule :

$$R = \frac{k}{n} \tag{2.24}$$

L'encodage se déroule de la manière suivante :

Soit le vecteur $\boldsymbol{u} = (u_1, u_2, .., u_k)$ contenant les bits d'information et le vecteur $\boldsymbol{v} = (v_1, v_2, .., v_n)$ correspondant au mot de code associé. La relation entre l'entrée $\boldsymbol{u}$ du codeur et sa sortie $\boldsymbol{v}$ est donnée ci-dessous :

$$\boldsymbol{v} = \boldsymbol{u}.\boldsymbol{G} \tag{2.25}$$

où $\boldsymbol{G}$ est la matrice génératrice du code de dimensions k, n.

À toute matrice génératrice $\boldsymbol{G}$, il est possible de calculer sa matrice de parité $\boldsymbol{H}$ de dimensions $(n-k,\ n)$ intervenant pour le décodage :

$$\boldsymbol{G}.\boldsymbol{H}^T = 0 \tag{2.26}$$

Au niveau du décodeur, on dit que la séquence reçu $\boldsymbol{r}$ est valide si :

$$\boldsymbol{S} = \boldsymbol{r}.\boldsymbol{H}^T = \boldsymbol{u}.\boldsymbol{G}.\boldsymbol{H}^T = 0 \tag{2.27}$$

où S est le syndrome.

Alors il existe un seul vecteur $\boldsymbol{u}$ correspondant à ce mot de code tel que :

$$\hat{\boldsymbol{u}} = \boldsymbol{r}.\boldsymbol{H}^T \tag{2.28}$$

La notion de syndrome est très importante pour renseigner le récepteur sur la présence d'erreurs de transmission. Si $\boldsymbol{S}$ est nul, il n'y a pas d'erreur dans le message reçu. Dans le cas contraire, il y a une ou plusieurs erreurs de transmission possiblement corrigibles par l'intermédiaire du syndrome. De plus amples informations sur le décodage des codes en bloc peuvent être trouvées dans [12] [13].

Pour la suite de ce mémoire, nous nous intéresserons seulement aux codes en blocs à faible densité de parité (LDPC) que nous détaillons dans le chapitre 3.

2.5 Représentation des codes sous forme de graphes

2.5.1 Graphes bipartis

En théorie des graphes, un graphe est dit biparti s'il existe une partition de son ensemble de sommets en deux sous-ensembles U et V telle que chaque arête (appellé aussi lien) ait une extrémité dans U et l'autre dans V. En d'autres termes, un graphe est biparti si et seulement si il ne contient pas de cycle de longueur impaire.

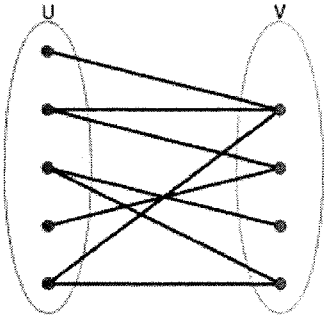


Figure 2.7: Graphe biparti

2.5.2 Le graphe de Tanner

Les graphes de Tanner [14] ont été découverts et utilisés par Michael Tanner comme un moyen pour créer des codes de grande dimension à partir d'autres codes de dimension plus petite en utilisant des algorithmes récursifs. Tanner a aussi montré les bornes inférieures sur les codes obtenus à partir de ses graphes.

Plusieurs définitions sur les graphes de Tanner seront nécessaires dans les prochains chapitres:

Définition 2.1:

Un graphe de Tanner est un graphe biparti, utilisé pour modéliser les équations d'un code correcteur d'erreur particulier. Ces graphes peuvent être utilisés dans l'encodage et le décodage de l'information.

Définition 2.2:

Les graphes de Tanner sont divisés en « subcode nodes » et en « digit nodes ». Pour les codes linéaires, les « subcode nodes » correspondent aux lignes de la matrice de parité ***H****. Les « digit nodes » représentent quand à eux les colonnes de cette même matrice.*

Un segment appelé « arête » ou « lien » connecte les nœuds de lignes avec ceux de colonnes si une valeur différente de 0 se trouve à l'intersection de la ligne et de la colonne, dans la matrice de parité.

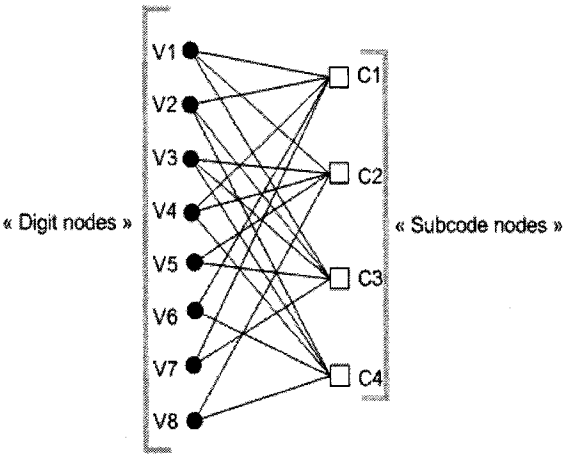
Enfin, la longueur du plus petit cycle présent dans le graphe est appelée « Girth ». Il s'agit d'un des facteurs les plus importants dans la performance des codes correcteurs d'erreur décodé sur graphe.

Pour conclure cette partie, un graphe biparti utilisant la notation de Tanner est présenté Figure 2.8 avec sa matrice de parité de taux R.

$$R = \frac{k}{n} = \frac{4}{8} = \frac{1}{2}$$

$$\begin{array}{c} \\ C1 \\ C2 \\ C3 \\ C4 \end{array} \begin{array}{c} \begin{array}{cccccccc} V1 & V2 & V3 & V4 & V5 & V6 & V7 & V8 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \end{array}$$

*Matrice de parité [**H**]*



Graphe de Tanner

Figure 2.8: Graphe biparti utilisant la représentation de Tanner [14] avec sa matrice de parité **H**

Le graphe de Tanner sera utilisé dans le chapitre 3 pour expliquer le fonctionnement des algorithmes de décodage sur graphe (Algorithme Somme-Produit) et dans la suite de ce mémoire comme support pour le comptage des cycles, puisque nous essayerons de mettre en relation les performances de décodage avec les cycles de petite longueur.

CHAPITRE 3

INTRODUCTIONS AUX CODES LDPC

3.1 Introduction

Ce chapitre est consacré à la présentation des codes à faible densité de parité LDPC utilisés dans la suite de ce mémoire. Le décodage par l'algorithme Somme-Produit sera traité et des exemples de construction de matrices de parité seront présentés. Mais tout d'abord, un rappel historique sur ces codes si particuliers est fait ci-après.

Les codes à faible densité de parité ont été découverts par Robert G. Gallager en 1960 lors de recherche pour sa thèse doctorat à MIT [2]. Il avait découvert un algorithme de décodage itératif qu'il avait utilisé avec cette nouvelle classe de codes. Il appela ces codes « Low density parity check » pour la simple raison que la matrice de parité avait besoin d'être très creuse pour être performante. Cependant, à cause de la complexité des calculs nécessaires au codeur et au décodeur, ces codes ont été oubliés pendant des années.

Ce n'est qu'en 1995 que Mackay et Neal redécouvrent ces codes et effectuent leur décodage par l'algorithme itératif « Belief Propagation » de Pearl [15]. Comme les codes Turbos [8] découverts en 1993, le décodage itératif est effectué sur un graphe. À partir de ce moment, les codes en graphe avec décodage itératif deviennent synonymes de très bonnes performances.

Par la suite, de nombreux travaux ont été réalisés sur de nouveaux algorithmes à décodages itératifs sur graphe et leurs points faibles ont été révélés par l'étude des performances d'erreur en présence de cycles de petite longueur [16].

3.2 Représentation graphique des codes en bloc

Les codes LDPC sont des codes linéaires obtenus à partir de graphes bipartis creux. Supposons que G est un graphe avec n nœuds à gauche (nœuds de variable) et r nœuds à droite (nœuds de parité). On obtient alors un code en bloc linéaire de longueur n. Les n bits du mot de code sont alors associés avec les n nœuds de variable.

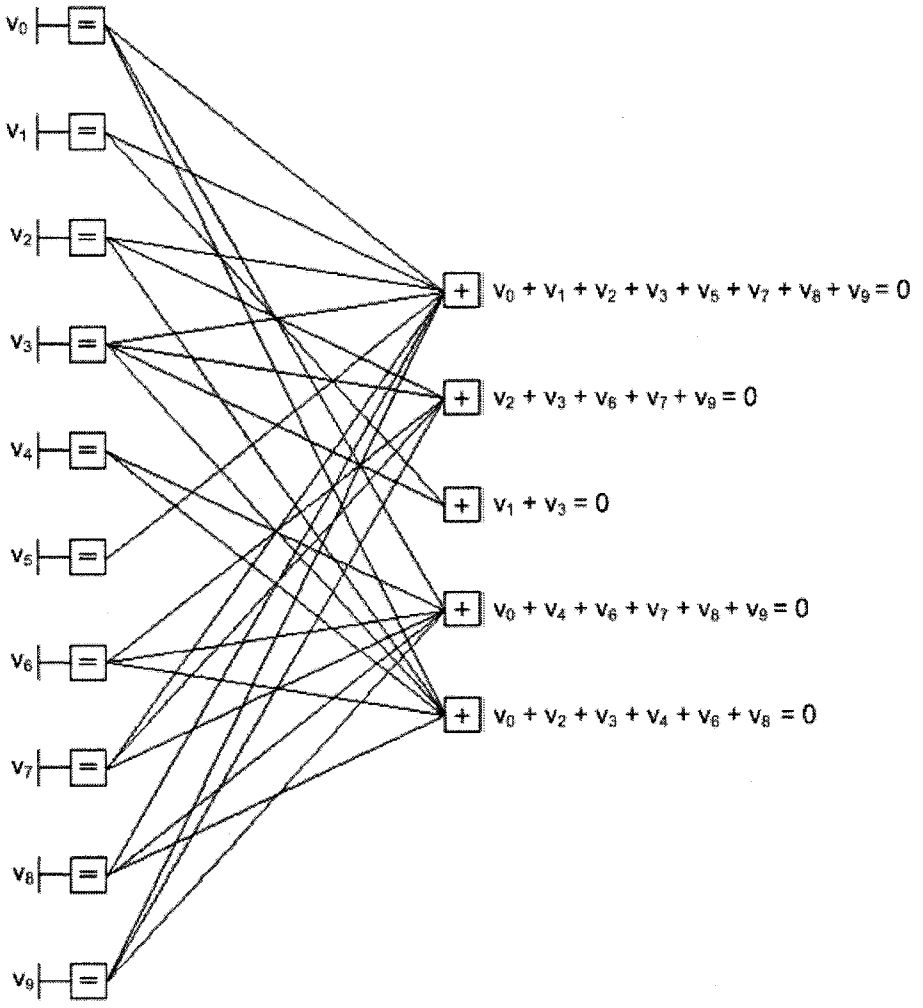
Les mots de code sont les vecteurs $\boldsymbol{c} = (c_1, \ldots, c_n)$ tel que pour tous les nœuds de parité, la somme de tous les voisins des nœuds de variable soit nulle. La Figure 3.1 illustre ceci.

La représentation sous forme de graphe est déterminée par la matrice associée. Soit **H**, une matrice binaire de dimension r, n dans laquelle les coordonnées (i, j) sont à 1 si et seulement si le i^{ème} nœud de parité est connecté au j^{ème} nœud de variable du graphe. Alors, le code LDPC défini par le graphe est l'ensemble des vecteurs $c = (c_1, \ldots, c_n)$ tel que :

$$\boldsymbol{H}.\boldsymbol{c}^T = 0 \tag{3.1}$$

La matrice **H** est alors appelée matrice de parité. N'importe qu'elle matrice binaire de dimension r, n peut être utilisée pour créer un graphe biparti avec n nœuds de variable et r nœuds de parité.

Par conséquent, n'importe quel code linéaire possède une représentation sous forme de graphe de Tanner. Cependant, tous les codes linéaires n'ont pas forcément une représentation par un graphe biparti creux. Si le graphe ou la matrice sont creux, alors le code est dit LDPC.



$$[H] = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Figure 3.1 Matrice de parité **H** (10,5) et sa représentation graphique sous forme de graphes de Forney[1] [17]

[1] La notation de Forney a le mérite d'être plus claire que celle de Tanner, puisqu'elle précise les opérations mathématiques nécessaires au décodage. Elle sera privilégiée dans la suite de ce mémoire.

3.3 Les classes de code LDPC

En 1962, lors de sa découverte, Gallager [2] a défini ses codes LDPC (n, j, k) comme des codes en blocs de longueur n avec j 1's sur chaque colonne et k 1's sur chaque ligne de la matrice de parité **H**.

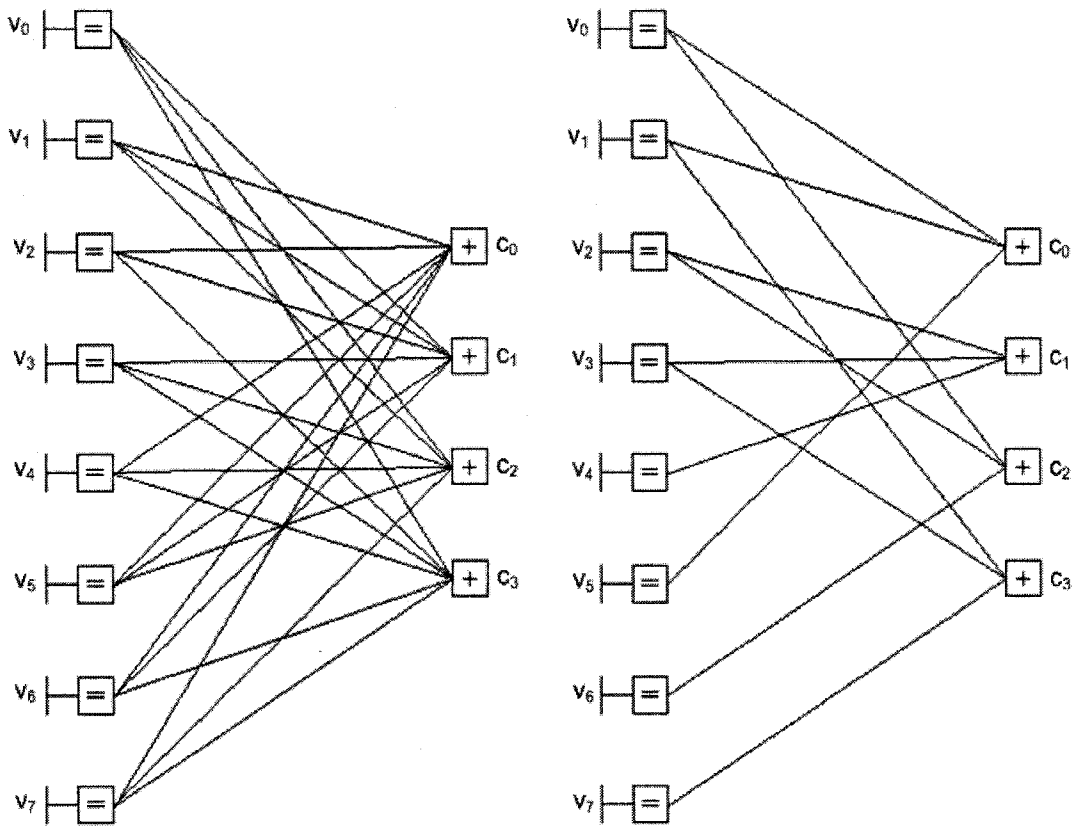
Les codes de Gallager ont donc un nombre fixe de 1's sur les lignes et les colonnes. D'un point de vue décodage, cela signifie que chaque bit est utilisé par des équations comprenant j symboles de parité et que chaque bit de parité est régi par une équation modulo-2 de k bits d'entrée. On retrouve le même nombre d'additions pour chaque équation. Cette classe des codes LDPC est appelée « Code LDPC régulier »

D'un autre point de vue, un code LDPC est dit régulier si les n nœuds de variable et les r nœuds de parité du graphe de Tanner sont de degré (j, k) constant avec $j \neq k$. Le taux du code doit alors vérifier :

$$R = \frac{n-r}{n} = 1 - \frac{j}{k} \tag{3.2}$$

La distribution du nombre de liens connectés à chaque nœud de variable (respectivement de parité) est donc constante. Ceci est équivalent à k 1's sur chaque ligne et j 1's sur chaque colonne comme mentionné précédemment. La Figure 3.2 illustre la différence entre un graphe d'un code régulier et d'un code irrégulier.

Plus de détails sur la construction des codes réguliers de Gallager sont disponibles en Annexe I.



Code régulier (n, j, k) = (8, 3, 6), R=1/2 Code irrégulier (n, r) = (8, 4), R=1/2

Figure 3.2 : Graphe de Forney [17] du code LDPC régulier de Gallager (8, 3, 6) et d'un code irrégulier (8, 4). Il s'agit de la représentation basée sur la matrice de parité.

Contrairement aux codes réguliers, les codes irréguliers [18] n'ont pas un nombre constant de 1's sur toutes les lignes et/ou toutes les colonnes.

Le nombre de 1's sur les colonnes est alors donné par la distribution des degrés $\lambda(x)$ et le nombre de 1's sur les lignes par la distribution des degrés $\rho(x)$ tel que décrit ci-après.

$$\lambda(x) = \sum_{i=2}^{dv} \lambda_i . x^{i-1} \tag{3.3}$$

$$\rho(x) = \sum_{i=2}^{dc} \rho_i . x^{i-1} \tag{3.4}$$

avec dv, le degré maximal des nœuds de variable,

dc, le degré maximal des nœuds de parité,

λ_i , ρ_i, le pourcentage de 1 dans les colonnes, respectivement lignes de poids i.

Par définition, Luby [18] posent $\lambda(1) = \rho(1) = 1$ et Γ comme le nombre de 1's dans la matrice de parité. Donc $\Gamma.\lambda_i$ représente le nombre total de 1's dans les colonnes de poids i. De plus, $\frac{\Gamma.\lambda_i}{i}$ est le nombre total de colonnes de poids i et $\sum_i \frac{\Gamma.\lambda_i}{i}$ est le nombre total de colonnes dans **H**.

Donc la proportion de colonnes ($\widetilde{\lambda_i}$) et de lignes($\widetilde{\rho_i}$) de poids i est la suivante :

$$\widetilde{\lambda_i} = \frac{\frac{\Gamma.\lambda_i}{i}}{\sum_j \frac{\Gamma.\lambda_j}{j}} = \frac{\frac{\lambda_i}{i}}{\sum_j \frac{\lambda_j}{j}} \; pour\ les\ colonnes \tag{3.5}$$

$$\widetilde{\rho_i} = \frac{\frac{\Gamma.\rho_i}{i}}{\sum_j \frac{\Gamma.\rho_j}{j}} = \frac{\frac{\rho_i}{i}}{\sum_j \frac{\rho_j}{j}} \; pour\ les\ lignes \tag{3.6}$$

D'autres méthodes de distribution des poids sur les nœuds existent pour obtenir des codes irréguliers, mais elles ne seront pas abordées dans ce mémoire.

3.4 Constructions de codes LDPC

Nous nous intéresserons dans cette partie à la construction de la matrice de parité **H** pour les codes LDPC. Nous décrivons ci-après quelques méthodes de construction de code LDPC. Deux catégories de techniques de construction existent dans la littérature: aléatoire ou déterministe [16].

Ce sont des techniques complexes, soumises à des règles difficiles à respecter. La mise en œuvre oblige que pour augmenter la longueur du plus petit cycle, la matrice soit plus creuse ce qui à pour conséquence de saboter la distance minimale et donc les performances. Cependant, la distance minimale augmente lorsque la matrice devient moins creuse, ce qui implique la création de petits cycles, car la dimension de **H** est finie. Par contre la présence de petits cycles a comme lourde conséquence de réduire la convergence des algorithmes de décodage, et donc de dégrader les performances d'erreur.

- Construction Aléatoire

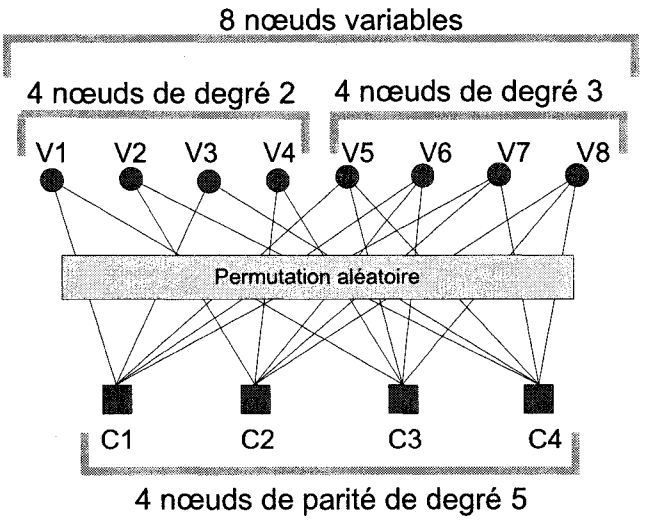
Il s'agit de la méthode la plus courante, utilisé par Gallager [2] en 1963 et reprise par Mackay et Neal [19] en 1995. La matrice de parité est alors le résultat de la concaténation et de la superposition de sous-matrices. Ces sous-matrices sont crées par des permutations aléatoires ou déterminées d'autres sous matrices qui ont un poids égal à 1.

Les codes réguliers ou irréguliers peuvent aussi être construits comme dans [20] où deux colonnes de nœuds sont crées. Chaque nœud est utilisé autant de fois que son degré l'y autorise. Un algorithme de permutation aléatoire établit alors les connexions entre les

deux colonnes de nœuds. Cette méthode est utilisé aussi bien sur les codes réguliers qu'irréguliers. Une illustration de ce principe de construction est donnée Figure 3.3.

Seul le degré attribué à chaque nœud varie avant l'exécution de la procédure. Cependant, les codes construits avec cet algorithme contiennent des cycles de longueur 4, ce qui n'est pas attrayant.

L'une ou l'autre de ces méthodes peut être contrainte afin d'éviter les cycles de longueur 4 et 6, par la vérification de l'absence de cycle de longueur 4 et 6 lors de l'ajout d'un 1 à l'endroit calculé par l'algorithme aléatoire. (i.e. « *Progressive Edge Growth* » (PEG)) [21].



$$[H] = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

Figure 3.3: Construction aléatoire de la matrice de parité **H** d'un code LDPC irrégulier $(8, 4)$, de taux $R = \frac{1}{2}$, utilisant la méthode de [20]

- Construction déterministe

Les méthodes de constructions aléatoires n'ont que très peu de contraintes pour la génération de matrices de parité. Elles ne garantissent pas par exemple, l'absence de petits cycles. Il est donc nécessaire d'ajouter des contraintes pendant la phase de génération aléatoire, ce qui augmente la complexité générale de l'algorithme de construction.

Les principaux problèmes sont la présence de petits cycles et la complexité du code. Pour remédier à cela, des méthodes de construction déterministe ont été proposées (i.e. PEG [21]) pour corriger certains problèmes particuliers. C'est par exemple le cas pour le standard WIMAX où l'on souhaite un bon compromis entre performance et complexité de codage et/ou décodage [22] [23].

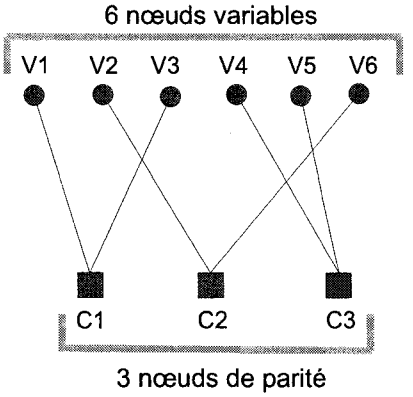
Ce sont des méthodes de géométrie finie qui permettent parfois de résoudre ces problèmes comme pour la détection de cycles de longueur 4, 6, 8 et 10 dans [24]. Les auteurs utilisent des formes géométriques prédéterminées pour analyser les matrices de parité et détecter des cycles de longueur donnée.

Enfin, une autre méthode pourrait consister à optimiser des matrices existantes en enlevant les liens participant à la formation de cycles. C'est cette technique que nous avons choisie de développer pour améliorer les performances des codes LDPC. Le chapitre 5 sera d'ailleurs partiellement consacré à l'étude de cette approche originale.

Il ne faut pas oublier qu'aujourd'hui, ces méthodes déterministes s'ajoutent souvent aux algorithmes de constructions aléatoires pour la génération des matrices de parité. Ces deux aspects dans la construction de codes LDPC sont complémentaires et nécessaires pour obtenir des codes performants.

3.5 Algorithme de décodage Somme-Produit

Le décodage des codes LDPC est rendu possible par l'utilisation d'un algorithme de décodage itératif optimal. Cependant cette optimalité est conditionnée par l'absence de cycles dans le graphe de Tanner, ce qui n'est pas le cas dans les codes couramment utilisés. L'optimalité est alors perdue en présence de cycles, mais de très bonnes performances sont tout de même obtenues. L'exemple de la Figure 3.4 montre une matrice de parité qui pourra être décodée de façon optimale.



$$[H] = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Figure 3.4: Matrice de parité **H** , d'un code (6, 3), de taux $R = \frac{1}{2}$, ne contenant aucun cycle et son graphe biparti associé.

De nos jours, un des algorithmes les plus répandus pour le décodage itératif des codes LDPC est connu sous le nom de Somme-Produit (SPA).

Cet algorithme a été utilisé par Gallager [2] pour le décodage de ses codes réguliers. L'algorithme propage des probabilités à travers les liens du graphe de Tanner pour réaliser le décodage. Un des aspects fondamental de cet algorithme est que le message envoyé par un nœud de variable V vers un nœud de parité C ne doit pas tenir compte du

message envoyé au cours de l'itération précédente, entre ces mêmes V et C. De même pour les messages envoyés des nœuds de parité aux nœuds de variable, comme illustré Figure 3.5.

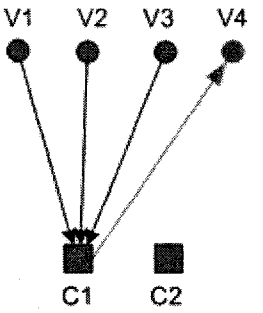
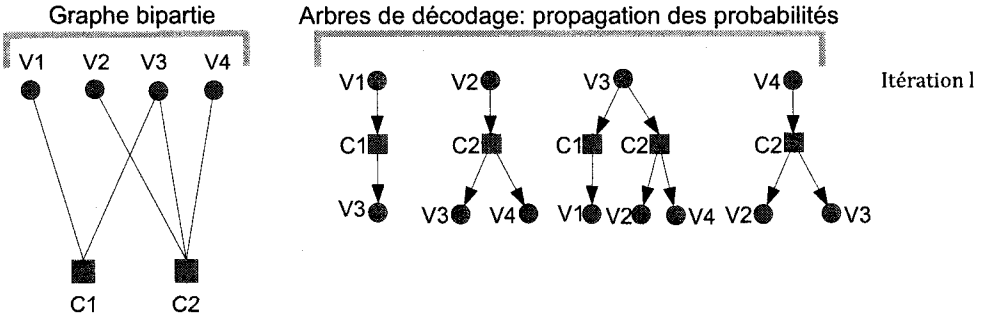


Figure 3.5: Liens à utiliser en C1 pour le calcul du message $m_{C1 \to V4}$

L'information envoyée des nœuds de variable Vi vers le nœud de parité C est composée de la probabilité que les Vi ait une certaine valeur sachant leurs valeurs au nœud de variable et de toutes les valeurs envoyées aux Vi au cours de leur dernière itération, des nœuds de parité incident aux Vi, excepté le nœud C.

De plus, le message de C vers V contient seulement la probabilité que V ait une certaine valeur sachant tous les messages envoyés à C, par tous les nœuds de variables incidents à C, sauf V.

La Figure 3.6 et les illustrations qui suivent tentent d'éclaircir tout cela.



$$[H] = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

Figure 3.6: Mise en évidence du passage des probabilités d'un nœud à l'autre à la $l^{ème}$ itération de l'algorithme.

Il est souvent plus difficile de travailler avec des probabilités. Il est donc coutume d'utiliser les rapports de vraisemblance dans le graphe de Tanner.

Soit la variable aléatoire binaire équiprobable x utilisée dans le calcul du rapport de vraisemblance :

$$L(x) = \frac{P[x = 0]}{P[x = 1]} \tag{3.7}$$

Soit une autre variable aléatoire y, non binaire, qui conditionne le rapport de vraisemblance.

$$L(x \mid y) = \frac{P[x = 0 \mid y]}{P[x = 1 \mid y]} \tag{3.8}$$

La règle de Bayes, pour le cas de variables aléatoires x équiprobables s'applique et donne :

$$L(x \mid y) = L(y \mid x) \tag{3.9}$$

Nous supposons que les variables aléatoires y_i, $i = 1, 2 \ldots l$, sont indépendantes entre elles. Ce qui nous permet d'écrire le logarithme de (3.9).

$$\ln[L\,(x \mid y_1, \ldots, y_l)] = \sum_{i=1}^{l} \ln L(x \mid y_i) \tag{3.10}$$

Supposons que les x_i soient des variables aléatoires binaires et que les y_i soient des variables aléatoires non binaires. L'expression (3.10) donne après développement [25] :

$$ln[L\,(x_1 \oplus \ldots \oplus x_l \mid y_1, \ldots, y_l)] = ln \frac{1 + \left(\prod_{i=1}^{l} \tanh \frac{\ln\,[L(x_i|y_i)]}{2}\right)}{1 - \left(\prod_{i=1}^{l} \tanh \frac{\ln\,[L(x_i|y_i)]}{2}\right)} \tag{3.11}$$

À la première itération, les nœuds de parité envoient sur tous les liens leur logarithme de vraisemblance, sachant la valeur observée. Pour cela, l'algorithme doit connaître le type de canal utilisé. Comme mentionné dans le chapitre 2, nous nous intéressons au BSC, avec une probabilité d'erreur p. Donc le premier message envoyé à tous les voisins d'un nœud de variable est :

$$ln(1-p) - ln(p) \quad Si\ la\ valeur\ du\ nœud\ de\ variable\ est\ 0 \tag{3.12}$$

$$ln(p) - ln(1-p) \quad Si\ la\ valeur\ du\ nœud\ de\ variable\ est\ 1 \tag{3.13}$$

Pour toutes les itérations suivantes, les nœuds de parité transmettent à leurs voisins le log de vraisemblance de l'équation (3.11). L'équation (3.10) est quant à elle utilisée lorsque les nœuds de variable envoient aux nœuds de parité leur log de vraisemblance, fonction des observations du canal et des logs de vraisemblance de l'itération précédente.

Le schéma (3.7) reprend les principes énoncés ci-dessus et effectue les premiers calculs lors d'une itération. Les nœuds où les calculs sont effectués sont mis en évidence.

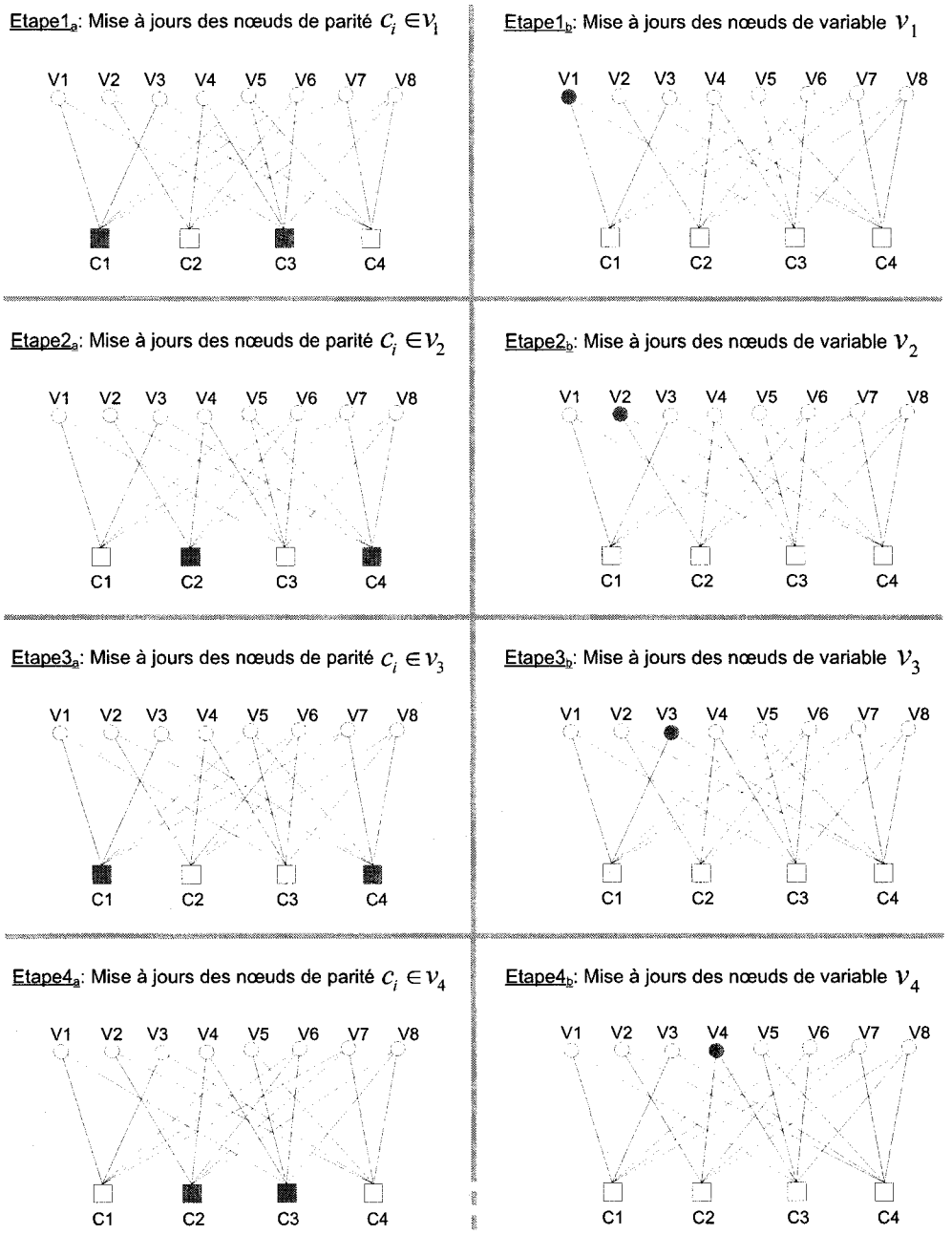


Figure 3.7: Mise en évidence des premières étapes de mise à jour par l'algorithme Somme-Produit lors d'une itération l quelconque.

Comme nous l'avons expliqué précédemment, le calcul du log de vraisemblance envoyé par le nœud de variable V vers le nœud de parité C tient compte de tous les logs de vraisemblance arrivés au nœud V à l'itération précédente sauf celui provenant du nœud C.

Soit $m_{VC}^{(l)}$, le message transmis à la $l^{\text{ème}}$ itération entre le nœud de variable V et le nœud de parité C.

Soit $m_{CV}^{(l)}$, le message transmis à la $l^{\text{ème}}$ itération entre le nœud de parité C et le nœud de variable V.

Voici alors les équations de mise à jour des messages :

$$m_{VC}^{(l)} = m_{VC}^{(0)} + \sum_{\substack{C' \in C_V \\ C' \neq C}} m_{C'V}^{(l-1)} \qquad pour\ l \geq 1 \tag{3.14}$$

$$m_{CV}^{(l)} = ln\left(\frac{1 + \left(\prod_{\substack{V' \in V_C \\ V' \neq V}} \tanh \frac{m_{V'C}^{(l)}}{2}\right)}{1 - \left(\prod_{\substack{V' \in V_C \\ V' \neq V}} \tanh \frac{m_{V'C}^{(l)}}{2}\right)}\right) \tag{3.15}$$

où C_V représente l'ensemble des connexions incidentes au nœud de variable V,

V_C l'ensemble des connexions incidentes au nœud de parité C,

et $m_{VC}^{(0)}$, le log de vraisemblance, fonction des valeurs reçues directement depuis le canal en V.

Cependant, lorsque l'algorithme n'a pas convergé à la dernière itération, la formule utilisée pour le calcul de $m_{CV}^{(l)}$ est modifiée. Pour améliorer très légèrement la performance d'erreur de l'algorithme, on prend maintenant en compte tous les liens incidents. Ceci se traduit par l'équation (3.16) [25]:

$$m_{CV}^{(l)} = ln\left(\frac{1+\left(\prod_{V'\in V_C}\tanh\frac{m_{V'C}^{(l)}}{2}\right)}{1-\left(\prod_{V'\in C}\tanh\frac{m_{V'C}^{(l)}}{2}\right)}\right) \tag{3.16}$$

Le décodage des codes LDPC avec l'algorithme Somme-Produit prend deux arguments essentiels [26]:

- Le nombre d'itérations maximum de l'algorithme (noté IT_{max})
- La valeur de la borne de confiance pour le log de vraisemblance $\ln L(x \mid y)$ proche de $\pm\infty$ à laquelle on admet que le nœud a été décodé correctement.

L'algorithme sera exécuté jusqu'à ce que tous les logs de vraisemblance aient convergés vers $\pm\infty$ ou que le nombre d'itérations maximal soit atteint.
En fonction de la valeur de vraisemblances obtenue au niveau des nœuds de variable, on décidera du bit décodé suivant les règles [26] :

- Si $\ln L(x \mid y) \to +\infty$, alors $P[x = 0 \mid y] = 1$ donc 0 est le bit décodé.
- Si $\ln L(x \mid y) \to -\infty$, alors $P[x = 1 \mid y] = 1$ donc 1 est le bit décodé.

En pratique, dès que l'on s'éloigne des très faibles SNR, le canal induit moins d'erreurs. Le nombre moyen d'itérations nécessaires au décodage en est alors réduit. A titre d'information, le nombre d'itérations moyen nécessaire pour décoder un bloc d'un code (500, 250) pour un BER à 10^{-5} est d'environ 4 itérations. Ceci à été mesuré sur les codes que nous avons simulés.
La procédure de décodage utilisant l'algorithme Somme-Produit que nous venons de détailler est schématisée Figure 3.8.

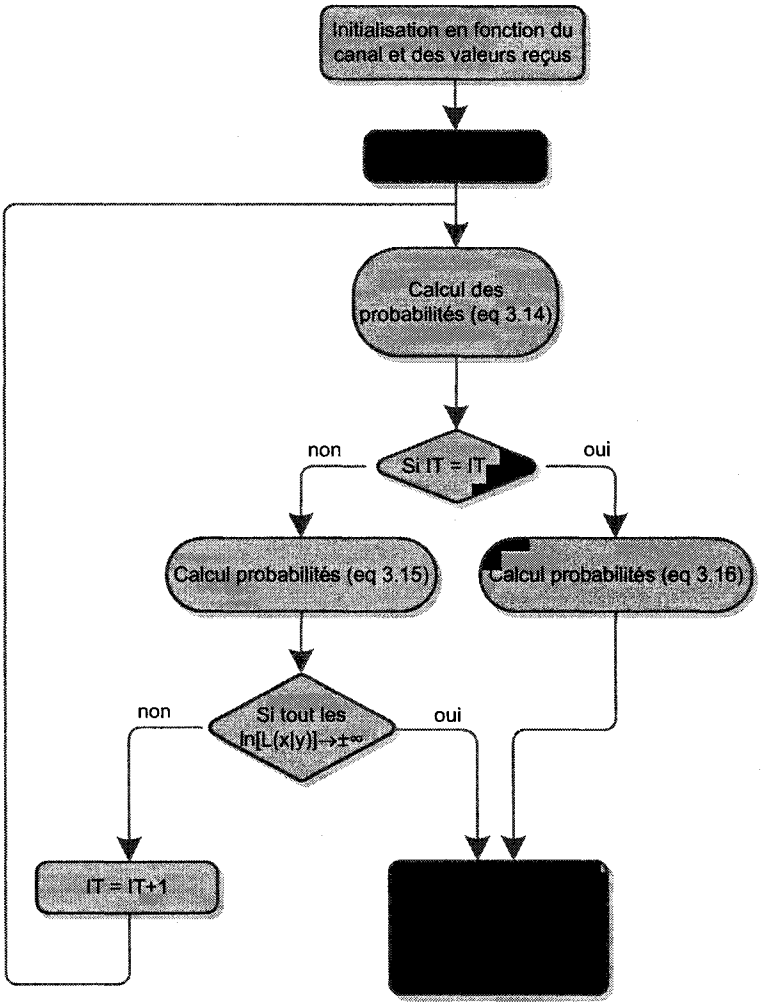


Figure 3.8: Schéma bloc de l'algorithme Somme-Produit utilisé dans ce mémoire

CHAPITRE 4:

THÉORIE DES CYCLES ET ALGORITHMES

4.1 Introduction

Le comptage exact des cycles et l'énumération des chemins dans les graphes sont connus pour être complexes. Le comptage nécessite la détermination de l'ensemble des chemins et ceci consomme beaucoup de ressources et de temps pour être effectué. Des algorithmes performants et intelligents, dans le sens où ils n'explorent pas tous les chemins, doivent être développés pour compter les cycles. Bien évidemment, il existe différentes longueurs de cycle. Ceux de longueur minimale peuvent être comptabilisés très rapidement contrairement aux plus grands qui posent souvent des problèmes.

Aucuns des algorithmes présentés ne s'inspirent de publications ou de méthodes connues, bien que la première des méthodes soit très évidente. Ceci est principalement la cause d'un manque de détails dans les deux publications consultées pour parvenir à reproduire les algorithmes [24][27].

Nous allons voir dans ce chapitre les difficultés rencontrés pour le comptage des cycles puis présenter deux algorithmes que nous avons spécialement développés et optimisés pour les graphes bipartis et enfin, nous conclurons sur des outils élaborés pour l'analyse des cycles.

4.2 Problématique du comptage de cycles

Le comptage des cycles minimaux relève des techniques d'exploration de graphes. La principale difficulté dans cette tâche est la préservation des ressources-mémoire. Pour bien situer le problème, nous nous proposons d'étudier l'exemple suivant.

Soit l'arbre de la Figure 4.1 où chacun des nœuds est relié à 1000 autres. À la profondeur 4, le graphe atteint $(10^3)^4 = 10^{12}$ chemins de longueur 4.

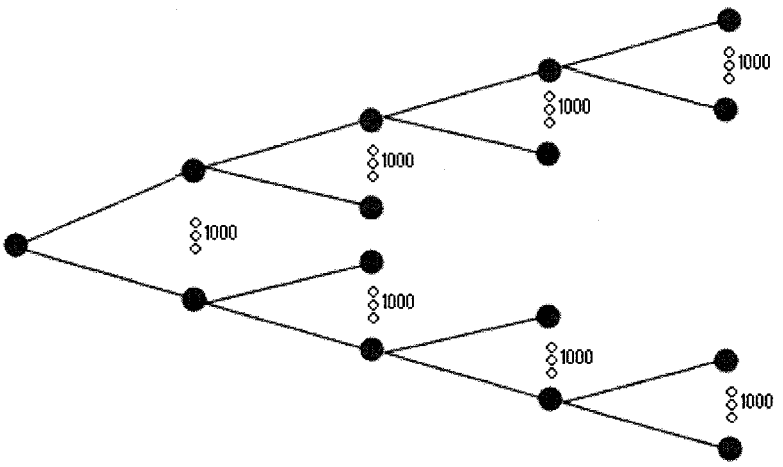


Figure 4.1: Illustration d'un arbre exploré jusqu'à la profondeur 4

Il peut être intéressant de chiffrer la mémoire minimale nécessaire à la mémorisation de l'ensemble des chemins possibles du graphe ci-dessus. L'utilisation de variable de type *Long* sur 32 bits est la plus appropriée (4 octets).

M = 4octets. (1000 + 1000^2 + 1000^3 + 1000^4) = 4.004 Téraoctets

Heureusement, les codes n'utilisent pas autant de liens et de nœuds. Cela n'empêche en rien une montée très rapide de la mémoire nécessaire au fur et à mesure que l'on rentre plus profondément dans le graphe.

Il est alors quasi impossible de mémoriser tous les chemins possibles dans un graphe de taille conséquente. Il faut donc parvenir à trouver un moyen de minimiser le nombre d'information à mémoriser pour compter les cycles les plus petits.

4.3 Algorithme d'exploration d'arbres

4.3.1 Comptage des cycles minimum

Une solution pour préserver les ressources est de compter les cycles sans mémoriser le chemin parcouru. En effet, il apparaît que seules la position actuelle et celle du nœud précédent sont nécessaires et suffisantes pour dénombrer les cycles minimaux. Cette propriété est d'ailleurs démontrée dans la prochaine partie.

Nous nous contenterons ici de montrer le fonctionnement de l'algorithme de comptage en prenant comme exemple le graphe biparti de la Figure 4.2.

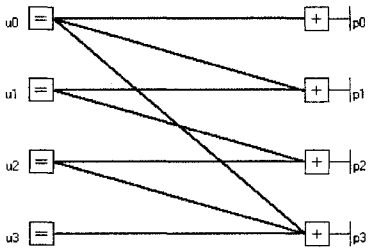


Figure 4.2: Graphe biparti avec un cycle de longueur N = 6
où les nœuds $u_i \in \mathcal{U}$ *et* $p_i \in \mathcal{V}$, $i = 0, 1, 2, 3$

Mais tout d'abord, quelques définitions sont nécessaires [27].

Définition 4.1 :

> *Un graphe* $G(\mathcal{W}, \mathcal{E})$ *est composé d'un ensemble non nul de nœuds* $\mathcal{W}$ *et de liens* $\mathcal{E}$, *se trouvant dans n'importe quel sous-ensemble de la paire:* $\{\{u, v\} : u, v \in \mathcal{W}, u \neq v\}$

Définition 4.2:

Un graphe $G(\mathcal{W},\mathcal{E})$ *est biparti avec des nœuds de la classe* $\mathcal{U}$ *et* $\mathcal{V}$ *si* $\mathcal{W} = \mathcal{U} \cup \mathcal{V}$, $\mathcal{U} \cap \mathcal{V} = \emptyset$ *et que chaque lien ait une extrémité dans* $\mathcal{U}$ *et l'autre dans* $\mathcal{V}$.

Définition 4.3 :

Un chemin de longueur N - 1 dans un graphe $G(\mathcal{W},\mathcal{E})$ *est une séquence de nœuds* $\alpha_1, \alpha_2, \dots, \alpha_N$ *où* $\alpha_i \in \mathcal{W}$ *et* $\{\alpha_i, \alpha_i + 1\} \in \mathcal{E}$, *pour tout* $\alpha_i \in [1, N]$.

Définition 4.4 :

Un chemin de longueur N dans $G(\mathcal{W},\mathcal{E})$ *est fermé si* $\alpha_1 = \alpha_{N+1}$. *C'est-à-dire si* α_1 *et* α_{N+1} *correspondent aux mêmes sommets* $v \in \mathcal{W}$.

Définition 4.5 :

Un chemin fermé de longueur N est un cycle si $\alpha_1, \alpha_2, \dots, \alpha_N$ *sont distincts. C'est-à-dire si* $\alpha_i \neq \alpha_j$ *pour tout* $i \neq j \in [1, N]$.

Étalons le graphe comme le ferait un algorithme d'exploration. Pour ce faire, certaines règles doivent être définies:

Règle1: Les retours en arrière sont interdits (pour mètre sous forme d'arbre).

Règle2: L'exploration commence par les nœuds $u_i \in \mathcal{U}$.

Règle3: Il existe au maximum un seul et unique chemin de longueur « 1 » entre un nœud de la classe $\mathcal{U}$ *et un nœud de la classe* $\mathcal{V}$.

Le graphe de la Figure 4.3 est alors obtenu en utilisant le graphe biparti de la Figure 4.2:

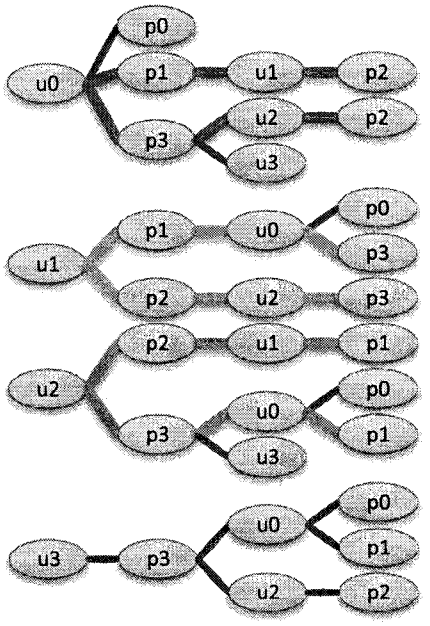


Figure 4.3: Graphe biparti étalé jusqu'à la profondeur $D = 3$
où trois cycles de longueur 6 sont mis en évidence

Si l'on regarde de plus près, le même cycle apparaît 3 fois. Seul l'ordre de la séquence est différent. Ceci donne lieu à la propriété 4.1.

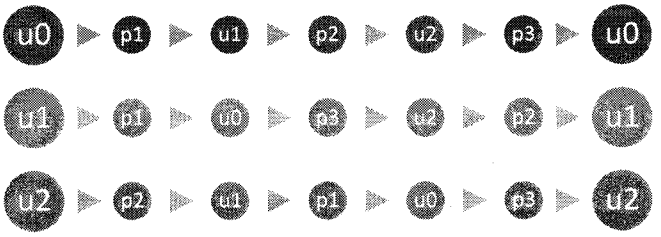


Figure 4.4: Mise en évidence de la présence du même cycle

Propriété 4.1:

Un cycle de longueur N dans un graphe biparti $G(\mathcal{W}, \mathcal{E})$ est composé de N/2 nœuds de gauche et N/2 nœuds de droite. Le cycle de longueur N sera donc comptabilisé N/2 fois par les algorithmes de comptage. Il faut en tenir compte pour donner le nombre exact de cycles d'un graphe.

Une démonstration par récurrence forte, démontrant qu'il n'est pas nécessaire de mémoriser l'intégralité du chemin pour le comptage de cycle minimum, est traitée dans la partie suivante.

4.3.2 Démonstration des conditions nécessaires pour les cycles minimaux

Cette partie est consacrée à la démonstration par récurrence forte, de la propriété de non-mémorisation du chemin pour le comptage de cycles minimaux.

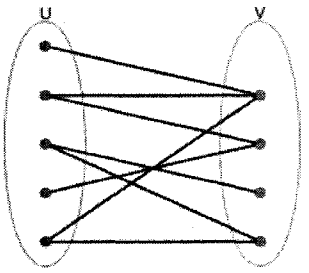


Figure 4.5: Graphe biparti

Soit $\mathcal{U}, \mathcal{V}$ l'ensemble des nœuds du graphe biparti $G(\mathcal{W}, \mathcal{E})$

L'exploration de ce graphe se fait dans le respect des règles précédemment énoncées, mais rappelées ci-dessous :

Régle1: *Les retours en arrière sont interdits*

Régle2: *L'exploration commence par les nœuds $u_i \in \mathcal{U}$*

Régle3: *Il existe au maximum un seul et unique chemin de longueur « 1 » entre un nœud de la classe $\mathcal{U}$ et un nœud de la classe $\mathcal{V}$.*

Cette démonstration suppose qu'il existe un ou plusieurs cycles, lorsque $N \to \infty$.

- *Initiation*: À la profondeur $D = 2$, les règles de construction confirment $B \neq B'$, $A \neq C' et\ A' \neq C$. Il faut seulement vérifier qu'il a la même origine et le même nœud final pour qu'il y ait un cycle (C'est à dire $A = A' et\ C = C'$). Ce n'est pas le cas.

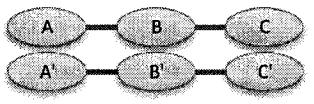


Figure 4.6: Récurrence forte (1)

- *Hérédité*: À la profondeur $D = 3$, les règles de construction confirment $C \neq C'$, $B \neq D' et\ B' \neq D$.

  Les propriétés vérifiées à l'étape précédente, c'est-à-dire pour $D = 2$, confirment que $B \neq B' et\ que\ A \neq C' et\ A' \neq C$.

  Il faut seulement vérifier qu'ils aient la même origine et le même nœud final pour avoir un cycle. Ce n'est toujours pas le cas.

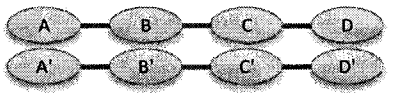


Figure 4.7: Récurrence forte (2)

- *Hérédité* [2]: A la profondeur $D = 4$, les règles de construction confirment $D \neq D'$, $C \neq E' et\ C' \neq E$

  Les propriétés vérifiées à l'étape précédente, c'est-à-dire pour $D = 3$ confirment que $B \neq B' et\ C \neq C' et\ que\ A \neq C' et\ A' \neq C$

  $et\ B \neq D'\ et\ B' \neq D$.

  Il faut seulement vérifier qu'ils ont la même origine et le même nœud final pour avoir un cycle. Ce n'est toujours pas le cas.

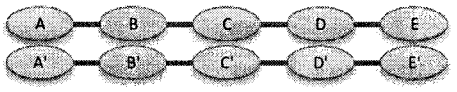


Figure 4.8: Récurrence forte (3)

- Si aucun cycle n'a été trouvé dans l'ensemble des étapes précédentes, c'est à dire $\nexists Cycle\ \forall\ D \in\ [2,\ N-1]$, alors par hérédité, un cycle de longueur minimal existe à l'étape présente D si et seulement si les deux chemins du graphe ont même nœud d'origine et même nœud final.

Cette démonstration prouve qu'il est inutile de mémoriser le chemin lors de l'exploration d'un graphe pour compter les cycles de longueur minimum. Elle donne lieu à la propriété 4.2 ;

Propriété 4.2:

Un chemin dans un graphe biparti $G(\mathcal{W},\mathcal{E})$ peut emprunter deux fois le même nœud si et seulement si la profondeur d'exploration est supérieure ou égale à la longueur du cycle minimum divisée par deux.

4.3.3 Comptage des cycles non minimaux de longueur 6 et 8

L'algorithme d'exploration de graphe décrit ci-dessus peut aussi être utilisé pour le comptage de cycles de longueur spécifique. Cependant, la propriété de « non-mémorisation du chemin emprunté » n'est plus valable, car il devient nécessaire de vérifier qu'un cycle de longueur inférieure n'est pas présent à l'intérieur de celui qui est comptabilisé. Ce cas particulier, appelé *lollipop* [27], est présenté à la Figure 4.9.

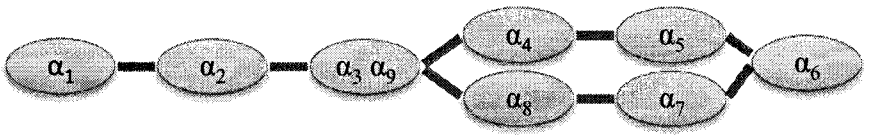


Figure 4.9: Lollipop (2, 6) de α_1 à α_3

Définition 4.6:

> *Le terme lollipop (m, N-m) désigne un chemin de longueur N où les nœuds* $\alpha_1, \alpha_2, \ldots, \alpha_N$ *sont distinct entre eux et* $\alpha_{N+1} = \alpha_{m+1}$. *Les cycles de longueur 2m sont donc des lollipop (0, 2m).*

L'algorithme d'exploration doit alors vérifier qu'aucun lollipop ne se trouve dans un cycle qu'il aurait découvert. Pour ce faire, une série de vérifications doit être effectuée. Ces vérifications dépendent de la profondeur d'exploration à laquelle l'algorithme est rendu. Les vérifications pour des cycles de longueur 6 et 8 sont présentées ci-après.

- Cycle de longueur 6 :

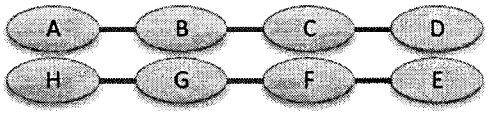


Figure 4.10: Cycle de longueur 6, formé par les deux chemins A...D et E...H

Le schéma de la Figure 4.10 montre deux chemins explorés jusqu' à la profondeur 3 et dont le regroupement forme un cycle de longueur 6. Ceci signifie que les nœuds A et H mais aussi D et E sont les mêmes. La figure 4.3 montre aussi des cycles de longueur 6. Vous pouvez vous y référer pour constater les égalités mentionnées ci-dessus.

Ceci donne lieu à la définition 4.7.

Définition 4.7:

Dans un graphe biparti $G(\mathcal{W}, \mathcal{E})$, exploré jusqu'à la profondeur 3, il existe un cycle de longueur 6 si et seulement si les nœuds des deux chemins vérifient les égalités suivantes :

$$A = H \ et \ D = E \tag{4.1}$$

$$B \neq G \ et \ C \neq F \tag{4.2}$$

Les autres vérifications ne sont pas nécessaires grâce aux règles d'exploration définies précédemment. Les relations (4.1) vérifient les extrémités des deux chemins et celle de (4.2) l'absence des deux cas de lollipop.

La vérification de ces relations est nécessaire et suffisante pour établir l'existence d'un cycle de longueur 6.

- Cycle de longueur 8 :

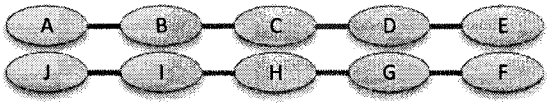


Figure 4.11: Cycle de longueur 8, formé par les deux chemins A...E et F...J

Définition 4.8:

Dans un graphe biparti $G(\mathcal{W}, \mathcal{E})$, exploré jusqu'à la profondeur 4, il existe un cycle de longueur 8 si et seulement si les nœuds des deux chemins vérifient les égalités suivantes :

$$A = J \ et \ E = F \ et \ A \neq E \tag{4.3}$$

$$B \neq I \ et \ D \neq G \ et \ C \neq H \tag{4.4}$$

$$B \neq G \ et \ I \neq D \tag{4.5}$$

Les relations (4.3) vérifient les extrémités des deux chemins, celle de (4.4) l'absence des six cas de lollipop et enfin celle de (4.5) le cas particulier d'un cycle de longueur 4 dans la boucle. Cette dernière est nécessaire si au moins un cycle de longueur 4 a été trouvé dans le graphe.

La vérification de ces relations est nécessaire et suffisante pour établir l'existence d'un cycle de longueur 8.

4.3.4 Schéma bloc des algorithmes

L'algorithme d'exploration de graphe utilisé, est le même pour les cycles de longueur N ou ceux de longueur minimale.

En effet, seules les vérifications nécessaires au niveau du comptage sont différentes, comme étudié dans la partie précédente.

Afin de limiter les problèmes de mémoire, une approche d'exploration en série a été envisagée. Au lieu d'explorer le graphe en démarrant l'ensemble des nœuds de départ simultanément, il apparaît intéressant de démarrer les nœuds de départ un par un. L'utilisation de la mémoire est alors divisée par le nombre de lignes de la matrice, mais les temps de calcul sont beaucoup plus longs. Cependant cette méthode permet de trouver le nombre de cycles de longueur 4, 6, et 8 pour de grands codes alors que la méthode d'exploration parallèle, est stoppée par le système d'exploitation pour cause de consommation trop importante des ressources-mémoire.

L'algorithme parallèle, qui est déjà le plus rapide des deux, déclare un processus (thread) par nœud de départ. Et donc, plus il y a de processeurs, plus le temps de calcul est réduit.

Les Figures 4.12 et 4.13 présentent les schémas blocs des deux algorithmes de comptage de cycle.

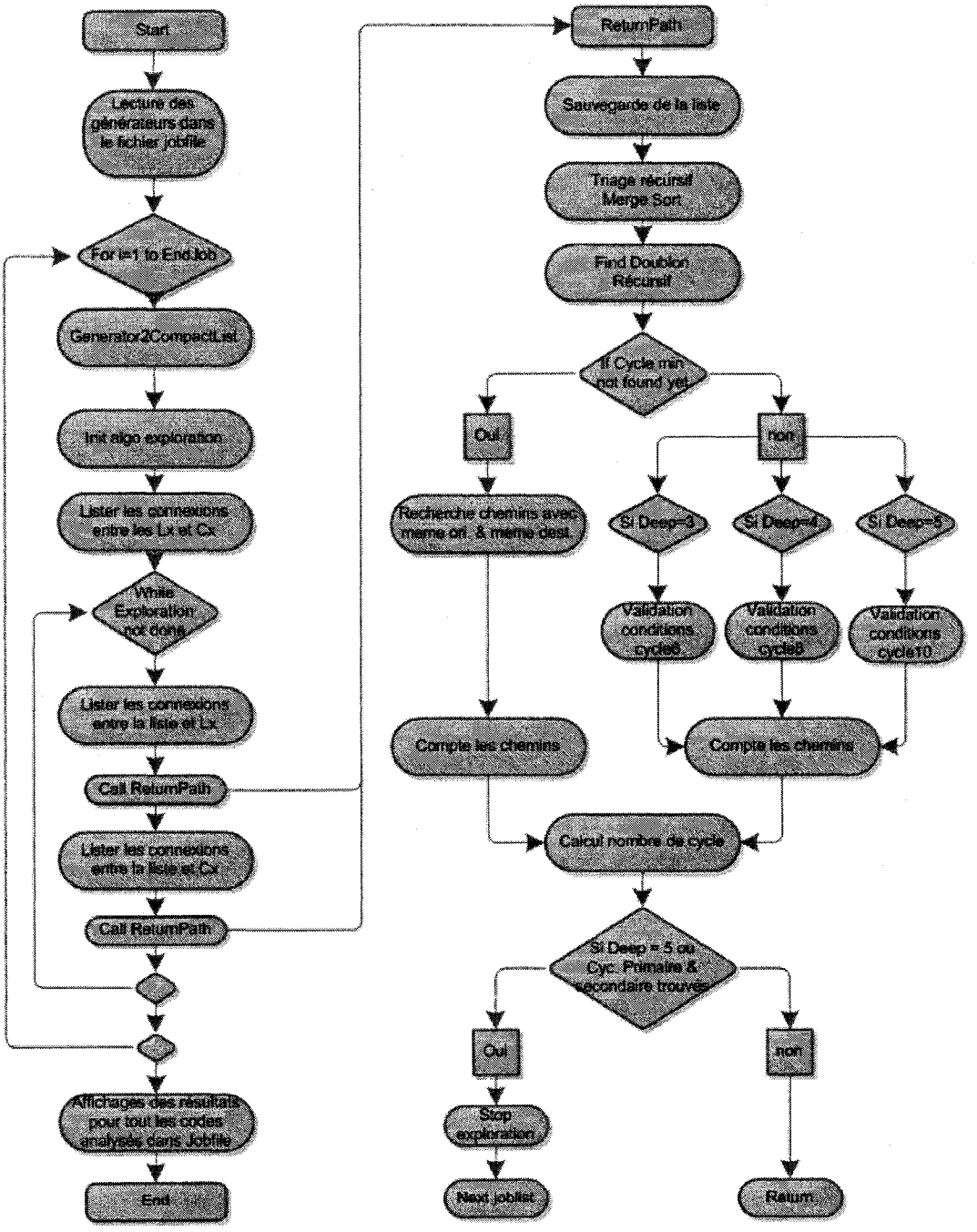


Figure 4.12: Schéma bloc de l'algorithme d'exploration parallèle du graphe pour le comptage de cycles. Cet algorithme consomme beaucoup de mémoire.

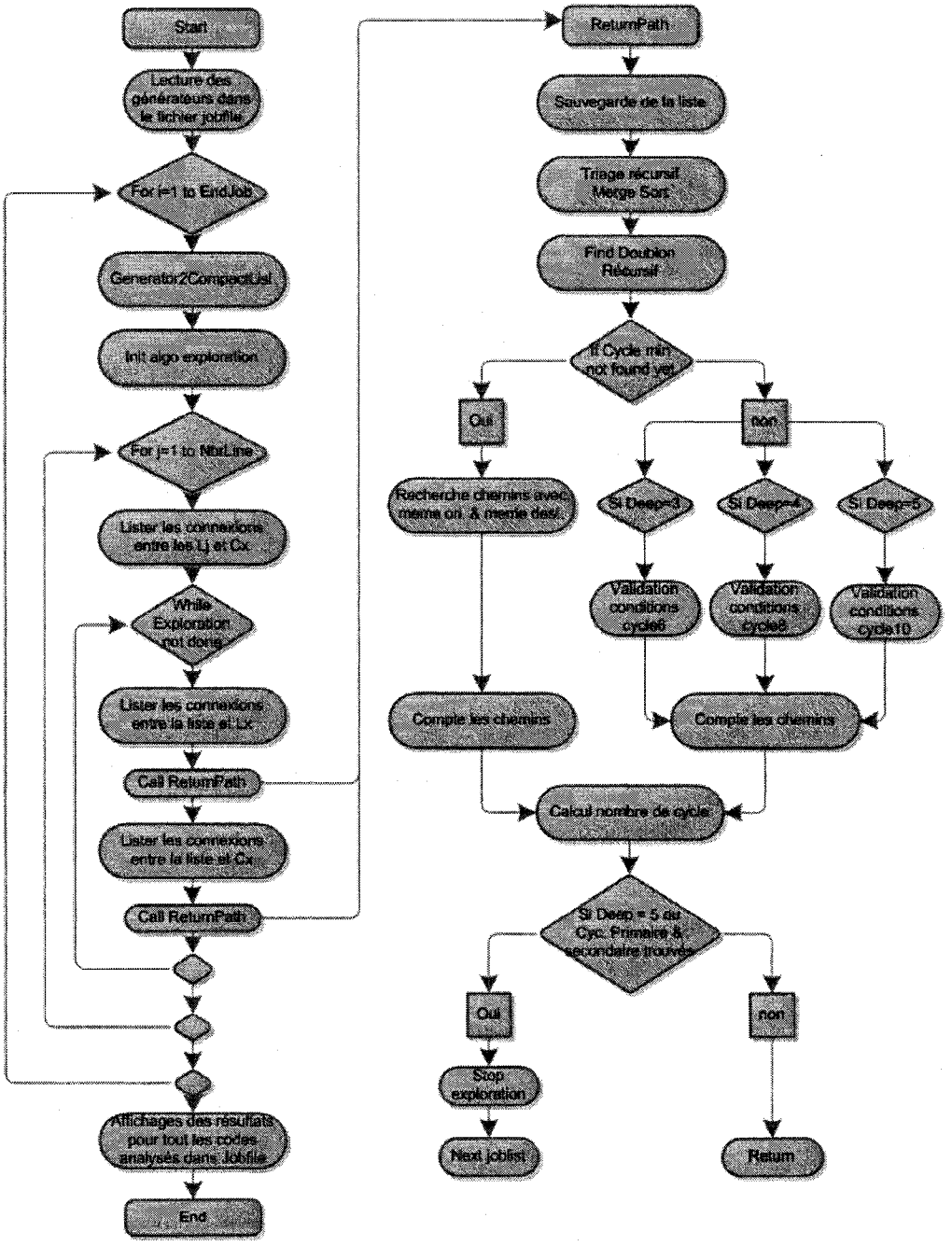


Figure 4.13: Schéma bloc de l'algorithme d'exploration série du graphe pour le comptage de cycles. Cet algorithme consomme peu de mémoire.

4.4 Algorithme d'exploration des graphes de Tanner

L'algorithme précédent, basé sur l'exploration d'arbres, n'est pas très performant. Au fur et à mesure que la profondeur d'exploration augmente, de plus en plus de mémoire et de temps sont nécessaires à l'algorithme. Il devenait utile de chercher une nouvelle méthode, plus efficace, pour diminuer ces problèmes de temps et de mémoire.

Cependant, certains objectifs ont changé après les premiers résultats obtenus avec l'algorithme d'exploration d'arbres, puisque le comptage des cycles non minimaux s'est avéré moins intéressant que prévu.

Pour parvenir à concevoir un nouvel algorithme performant, il fallait changer la méthode d'exploration. C'est pour cela que le comptage des cycles devait se faire sur un graphe biparti.

La partie suivante explique notre méthode, que nous avons conçue et développée pour atteindre ces objectifs.

4.4.1 Recherche des cycles dans un graphe de Tanner

Pour compter les cycles de longueur minimum dans un graphe de Tanner, nous nous sommes inspirés des algorithmes de décodage. Le nœud de départ Nd_J propage sur ses liens une étiquette qui va se déplacer de nœud en nœud. Lorsque plusieurs étiquettes atteignent à la profondeur D un même nœud, l'algorithme déduit la présence d'un ou plusieurs cycles, passant par ce nœud. Il utilise alors la formule 4.6 pour calculer le nombre de cycles N_i de longueur $2D$ passant par ce nœud i.

$$N_i = \frac{m_i(m_i - 1)}{2} \quad \textit{(Somme de Gauss)} \tag{4.6}$$

avec m_i le nombre de marques (étiquettes) sur le nœud i.

Une fois tous les marquages vérifiés et les N_i calculés, le nombre de cycles $Ncyc_{2D,\ Nd_j}$ à la profondeur $2D$, utilisant le nœud de départ Nd_j peut-être calculé.

$$Ncyc_{2D,\ Nd_j} = \sum_{i \in u} N_i \tag{4.7}$$

Il est nécessaire de calculer le nombre de cycles $Ncyc_{2D,\ Nd_j}$ à la profondeur du cycle minimum, pour tous les nœuds de départ $Nd_j \in \mathcal{U}$ ou $\in \mathcal{V}$. Les nœuds de départ doivent absolument appartenir au même sous-ensemble du graphe biparti.
Si les $Nd_j \in \mathcal{U}$, un comptage suivant les nœuds de variable est effectué.
Si les $Nd_j \in \mathcal{V}$, un comptage suivant les nœuds de parité est effectué.

Une fois l'ensemble des $Ncyc_{2D,\ Nd_j}$connu à la profondeur D du cycle minimal, il est possible de connaître le nombre de cycles N_{2D} de longueur $2D$ qui jalonnent le graphe.

$$N_{2D} = \frac{1}{D} \sum_{i \in u} Ncyc_{2D,\ j} \tag{4.8}$$

L'algorithme qui vient d'être énoncé est schématisé à la Figure 4.14.

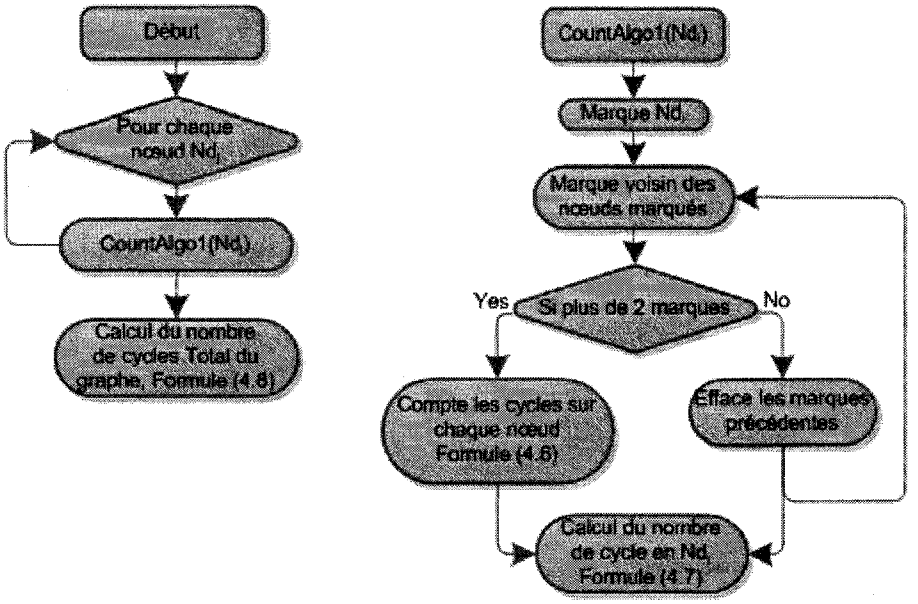


Figure 4.14: Schéma bloc de l'algorithme de comptage des cycles minimaux

La présence de lollipop empêche le comptage des cycles de longueur supérieure au cycle minimum. Cependant, il est possible de compter les cycles à la profondeur $D + 1$ en complexifiant les étiquettes. Mais cette méthode n'a pas été implémentée.

Les Figure 4.15 et 4.16 justifient l'utilisation de la formule (4.6) et illustre le système de propagation d'étiquette définie plus haut.

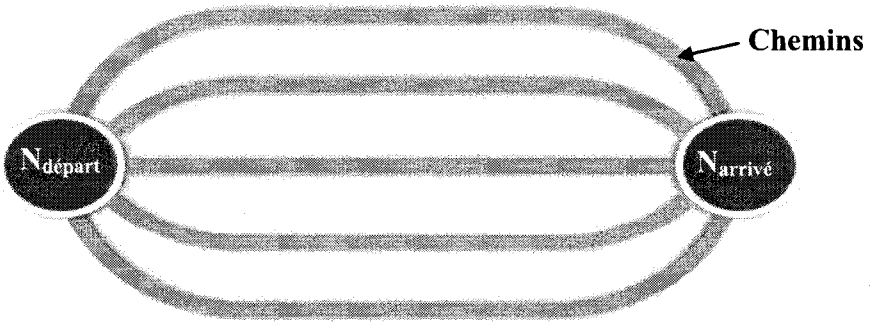


Figure 4.15: Exemple propageant 5 étiquettes pour justifier la formule (4.6)

La Figure 4.15 montre l'existence de cinq chemins entre un nœud de départ et un autre nœud du graphe. Il est alors possible de compter visuellement tout les cycles minimaux passant par ce nœud de départ.

On trouve 10 comme résultat de la somme : $4 + 3 + 2 + 1 + 0$.

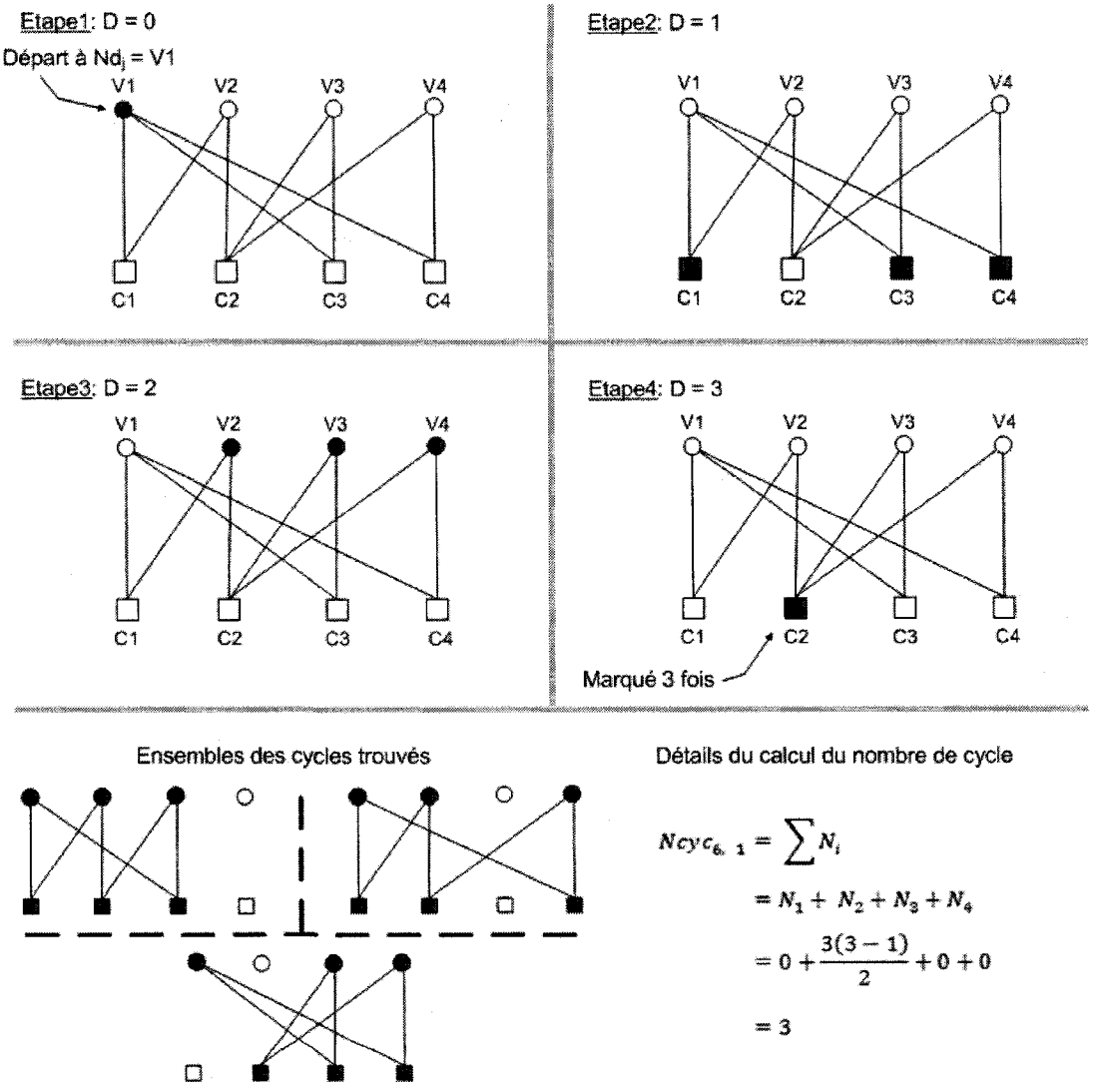


Figure 4.16: Propagation des étiquettes à partir du nœud de départs $Nd_1 \in \mathcal{U}$ où 3 6-cycles (3 cycles de longueur 6) ont été trouvés

4.4.2 Complexité des algorithmes

La complexité de cet algorithme a été analysée, obtenant des performances au niveau mémoire et vitesse d'exécution très concluants. Les temps de calcul, pour un ordinateur comparable à celui utilisé pour ce mémoire, peuvent être calculés avec les formules suivantes. Ces formules expérimentales ont été obtenues grâce à la mise en place de marqueur temporel dans le code source des deux programmes.

$$Calcul_{Algo1}(\mu s) = \left(0{,}128\,.\,Nbr_{Nd}^2 - 56{,}13\,Nbr_{Nd}\right).\frac{Nbr_{LiensParNoeud}}{3}.\frac{Deep^2}{2} + Cte \qquad (4.9)$$

$$Calcul_{Algo2}(\mu s) = \left(0{,}037\,Nbr_{Nd}^2.Nbr_{LiensParNoeud} + 0{,}589\,Nbr_{Nd}\right).\frac{Deep}{2} \qquad (4.10)$$

avec Nbr_{Nd} le nombre de nœuds de départ, et $Deep$ la profondeur explorée.

Code ID	Complexité	Code g = 4	Code g = 6
Algo1	$O(Nbr_{Nd}.Deep)^2$	0.253 sec	1.216 sec
Algo2	$O(Nbr_{Nd})^2$	0.111 sec	0.173 sec

Tableau 4.1: Comparaisons de la complexité et des temps de calcul des deux algorithmes pour deux codes réguliers de Gallager (500, 1000, 3) avec un cycle minimum à 4 et à 6.

Même si la complexité est en $O(n)^2$ pour les deux algorithmes, une constante et un impact supérieur à la profondeur, se rajoutent dans le premier cas.

La constante varie avec le temps d'allocation de la mémoire, et la profondeur d'exploration augmente significativement avec le nombre de chemins à analyser.

Ceci n'est pas vrai pour le deuxième algorithme qui occupe une mémoire et un nombre de chemins constant. Cela lui permet d'avoir une complexité linéaire avec la profondeur explorée.

4.5 Comptage élaboré des cycles

Nous introduisons dans cette partie trois méthodes pour déterminer le nombre de cycles de longueur minimale parcourant chaque nœud et lien d'un graphe de Tanner. Ces méthodes sont destinées à collecter plus d'information que les méthodes classiques [28], qui ne renvoient que le nombre de cycles de longueur *g, g+2 et g+4*.

Nous présentons dans les paragraphes suivants trois algorithmes. Le premier compte le nombre de cycles minimaux pour tous les nœuds N_i du sous-ensemble $\mathcal{U}$ et $\mathcal{V}$ du graphe biparti, le deuxième effectue la même chose pour chaque lien du graphe, et le dernier recherche la longueur du plus petit cycle passant par un nœud ou un lien du graphe de Tanner.

4.5.1 Comptage des cycles minimaux sur les nœuds

Ce premier algorithme compte le nombre de cycles de longueur minimale utilisant un nœud donné du graphe de Tanner. L'algorithme utilisé est celui de la partie précédente qui propage des étiquettes sur les liens du graphe biparti. L'exemple de la Figure 4.17 montre les distributions des cycles de longueur minimale, pour les nœuds $N_i \in \mathcal{U}$ à gauche et les nœuds $N_i \in \mathcal{V}$ à droite, de la matrice de parité (4.11).

$$\boldsymbol{H} = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.11}$$

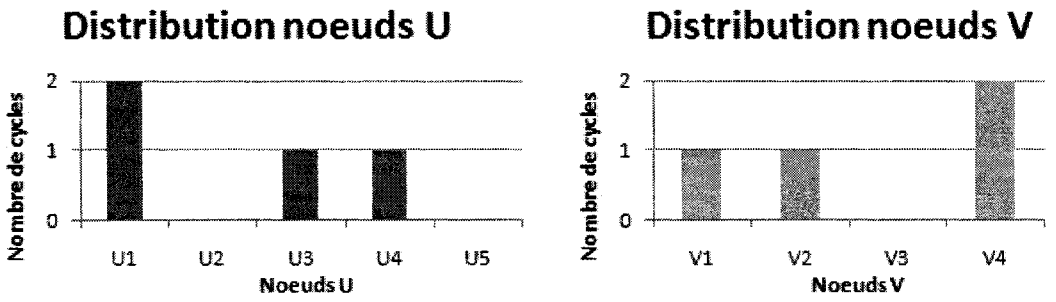


Figure 4.17: Distribution des cycles de longueur minimaux sur les nœuds $N_i \in \mathcal{U}$ et $N_i \in \mathcal{V}$ de la matrice de parité (4.11)

4.5.2 Comptage des cycles minimaux sur les liens

L'algorithme de la Figure 4.14 permet de déterminer le nombre de cycles de longueur minimaux par nœud. Cependant, des informations plus précises sur le cycle minimum d'un code correcteur d'erreur peuvent être trouvées en comptant le nombre de cycles partageant le même lien. Pour cela un comptage des cycles à partir d'un des deux nœuds du lien est nécessaire. Ce comptage est effectué avec et sans le lien concerné. La formule (4.12) donne le nombre de cycles parcourant le lien.

$$Ncyc_{lien\,(i,j)} = (Ncyc_{noeud\,i})_{lien(i,j)\,présent} - (Ncyc_{noeud\,i})_{lien(i,j)supprimé} \qquad (4.12)$$

Le schéma bloc de la Figure 4.18 reprend les principes que nous venons d'énoncer.

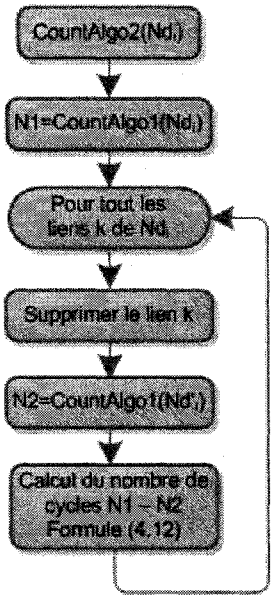
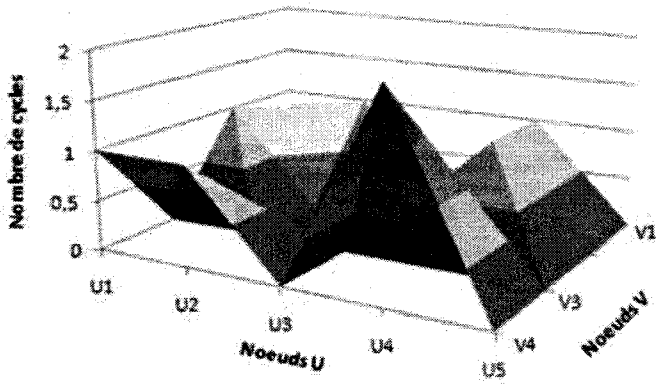


Figure 4.18: Schéma bloc de l'algorithme de comptage des cycles minimums sur chaque lien, d'un même nœud de départ Nd_j.

Une fois les liens du sous-ensemble $\mathcal{W}$ analysés par l'algorithme, le nombre de cycles partageant un même lien est obtenu.

Un lien est représenté par deux coordonnées (i,j), correspondant aux nœuds des sous-ensembles $N_i \in \mathcal{U}$ et $N_j \in \mathcal{V}$ qu'il relie. Aussi, le nombre de cycles minimaux comptabilisés sur ce lien peut être défini par les mêmes coordonnées.

Alors une matrice des cycles minimaux peut être construite. En reprenant la matrice de parité (4.11), la distribution des cycles sur les liens est obtenue à la Figure 4.19.



$$M = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 4.19: Visualisation de la matrice des 4-cycle[1] des liens $\boldsymbol{M}$ du graphe biparti défini par la matrice (4.11)

Ces résultats permettent d'identifier les liens engagés dans un grand nombre de cycles. Cet algorithme est utilisé dans le chapitre 5 pour identifier les points faibles des matrices de parité et dans le chapitre 6 pour l'analyse des codes S-CSO2C-WS.

4.5.3 Comptage des cycles propres à un nœud ou à un lien

Après avoir compté le nombre de cycles minimaux sur les liens et nœuds du graphe de Tanner, nous introduisons dans cette partie un algorithme qui recherche la longueur du plus petit cycle passant par un nœud ou un lien du graphe de Tanner.

Il utilise pour cela une version modifiée de l'algorithme de propagation d'étiquettes qui, au lieu de compter les cycles minimaux, enregistre pour un nœud ou un lien donné, la longueur du plus petit cycle.

Comme pour les deux autres algorithmes de la section 4.5, les résultats se présentent sous forme de vecteur, dans le cas des nœuds, ou de matrice, dans le cas des liens. Cet algorithme est utilisé dans le chapitre 5 pour le calcul des cycles minimaux moyens associés à un code.

[1] La terminologie « X-cycle » signifie « nombre de cycle de longueur X ».

CHAPITRE 5:

LES CYCLES ET LES CODES LDPC

5.1 Introduction

Les performances des codes LDPC avec un décodage itératif ont surpassé celles de tous les codes connus. Aujourd'hui, les algorithmes de décodage comme le Somme-Produit sont déjà optimisés et quasi-optimaux pour le décodage des graphes avec cycles. Il semble maintenant qu'aucune amélioration significative ne soit possible par l'amélioration ou la création d'algorithmes de décodage [16].

Le travail sur les matrices de parité semble être la clef pour améliorer les performances d'erreur.

Depuis plusieurs années, l'ensemble des chercheurs s'accorde à dire que l'importance des cycles est prépondérant sur les performances observées. Leur étude de manière poussée est nécessaire pour comprendre leur rôle réel dans la dégradation des performances d'erreur. Des outils et des méthodes d'analyse pointus doivent être créés pour atteindre ces objectifs.

Nous proposons dans ce chapitre, qui présente notre contribution la plus importante, de définir des méthodes d'analyse basées sur les algorithmes du chapitre 4 pour étudier et définir le lien qui existe entre les cycles et la dégradation des performances d'erreur. Nous verrons entre autres comment il est possible d'améliorer les performances d'un code existant avec notre algorithme PER[1] (*Progressive Edges Reduction*) qui supprime les points faibles du code. Nous présenterons aussi une méthode originale pour sélectionner les meilleurs codes des algorithmes de recherche exhaustive. Enfin, nous aborderons le paradoxe du besoin de cycles dans le décodage de codes implémentables.

---

[1] La méthode d'amélioration de matrice de parité PER à été conçue et implémentée pour ce mémoire. Elle porte sont nom en référence à la méthode PEG qui elle au contraire, ajoute des liens au lieu de les enlever. L'idée de cette méthode est un hasard puisque à l'origine, nous cherchions à classer des codes LDPC par performance d'erreur.

5.2 Méthode d'optimisation de matrice de parité (PER)

Nous avons présenté dans le chapitre précédent un algorithme de comptage de cycles sur les liens et nœuds d'un graphe de Tanner. Cet outil va permettre de comprendre plus en détail l'impact des petits cycles sur les performances d'erreur.

Comme il a été précédemment expliqué, cet algorithme identifie combien de fois un lien particulier du graphe de Tanner est utilisé dans la formation de cycles minimaux, et enregistre cette information dans ce que nous appelons « *une matrice des cycles* », noté ***M***. Nous pensions qu'avec une étude poussée de cette matrice des cycles, il serait possible de trouver un paramètre récurent pour prévoir quel code parmi plusieurs devait être le meilleur. Malheureusement, cette étude n'a pas été fructueuse. Il n'était pas possible d'obtenir un classement des codes par leurs performances d'erreur. Mais cela ne signifie pas pour autant que c'est impossible.

Notre idée était que cette matrice des cycles contient les points faibles quantifiés[1] de la matrice de parité d'un code et qu'en conséquence l'analyse de ces points faibles permet d'identifier la matrice la plus performante parmi plusieurs construites avec les mêmes propriétés. Cependant, il est indéniable que cette matrice des cycles contient bien les points faibles du code. C'est cela qui nous a donné l'idée d'exploiter ces points faibles pour améliorer les performances de codes LDPC. Pour atteindre cet objectif, il suffit d'éliminer les points faibles de la matrice de parité et les performances d'erreur à haut SNR s'améliorent d'elles-mêmes.

Cependant, l'expérience nous a appris qu'il fallait prendre quelques mesures préventives pour limiter la dégradation des performances due à la suppression d'un lien.

Voici quelques règles à respecter :

- Ne pas déconnecter un nœud du reste du graphe.
- Supprimer en priorité les liens impliqués dans le plus grand nombre de cycles en espérant une diminution importante de ces derniers.
- Se déplacer aléatoirement dans la matrice pour la suppression des liens.

[1] Le mot quantifié est utilisé car l'algorithme compte le nombre de cycles passant par un même lien. Donc si la matrice des cycles contient à un endroit un chiffre élevé, cela signifie que le lien concerné est engagé plus que les autres dans la formation de cycles.

Bien évidemment, la première mesure sert à ne pas ignorer un bit arrivant du canal.
La deuxième sert à limiter au maximum le nombre de liens à supprimer, car la suppression des liens réduit les distances entre certains mots de code et ainsi dégrade les performances d'erreur.
Enfin, la troisième règle est utile pour bien répartir la suppression des liens dans la matrice de parité.
L'algorithme d'optimisation d'un graphe de Tanner existant est donné Figure 5.1.

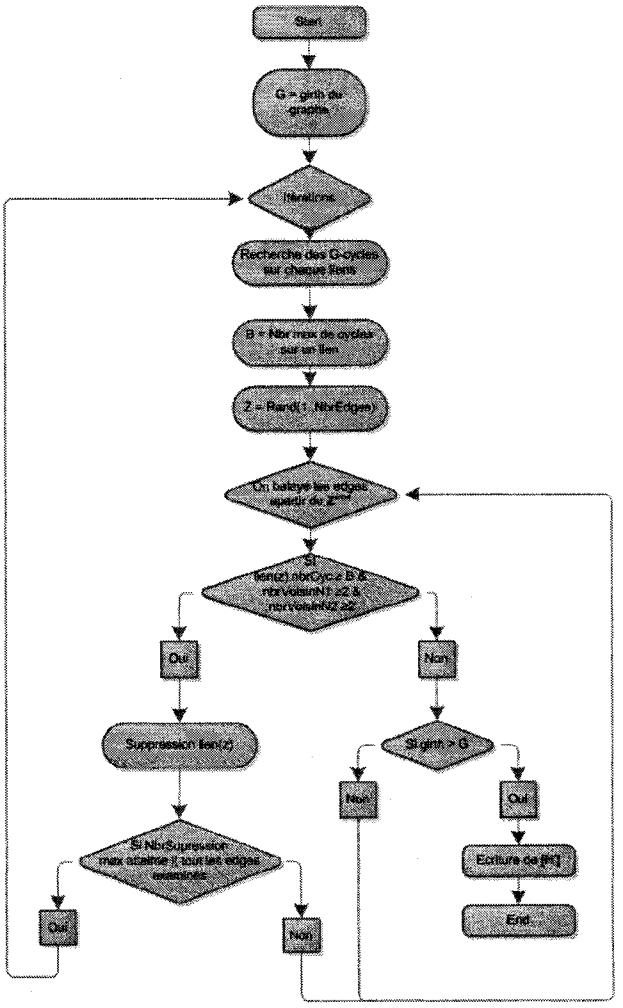


Figure 5.1: Schéma bloc de notre algorithme d'optimisation de matrice de parité PER (*Progressive Edges Reduction*) utilisant certaines fonctions développées au chapitre 4.

Illustrons cette méthode par un exemple :

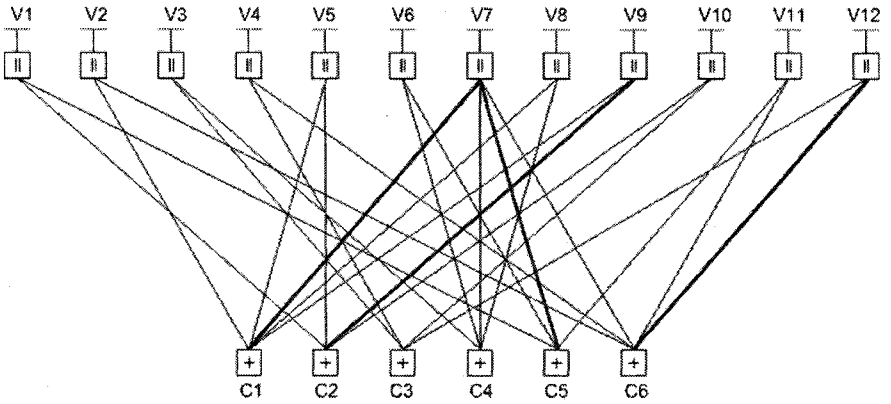
- La première étape consiste à générer une matrice de parité **H**, régulière ou irrégulière, avec une méthode de construction aléatoire. Pour cela, nous avons choisi d'utiliser l'algorithme de Radford M. Neal [3] qui génère un code LDPC régulier non systématique.
- Une fois la matrice **H** obtenue, il ne reste qu'à appliquer l'algorithme d'optimisation qui va supprimer un par un les liens utilisés dans la formation de cycles minimaux. Ces liens vont être analysés afin de déterminer leur ordre de suppression. Ceux responsables dans un grand nombre de cycles seront éliminés en priorité. La liste définissant l'ordre de suppression des liens est mise à jour après chaque suppression. Ainsi, le nombre d'éléments supprimés est réduit au minimum. Ce protocole permet de résoudre de manière quasi optimale ce problème qui est NP complet ; à savoir, supprimer le nombre minimum de liens dans le graphe de Tanner d'un code pour obtenir un nouveau code avec un cycle minimum supérieur au premier. Après l'exécution de cet algorithme, le cycle minimum g_H du code devient :

$$g_{H'} \geq g_H + 2 \tag{5.1}$$

La matrice de parité quasi-régulière **H'** est alors obtenue. La légère irrégularité étant causée par la suppression des 1's dans la matrice. Un exemple des matrices **H** et **H'** est donné à la Figure 5.2

Enfin, il ne reste plus qu'à simuler les deux codes pour observer l'amélioration obtenue. La Figure 5.3 montre les performances d'erreur avant et après l'optimisation de la matrice.

Des résultats complémentaires, effectués sur d'autres codes, sont présentés en Annexe II.



$$H' = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & \mathbf{0} & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \mathbf{0} & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & \mathbf{0} & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & \mathbf{0} \end{bmatrix}$$

Figure 5.2: Pour supprimer les 6 4-cycles de la matrice **H**, l'algorithme à supprimé quatre liens. **H'** est la matrice obtenue. Les liens supprimés sont en rouge.

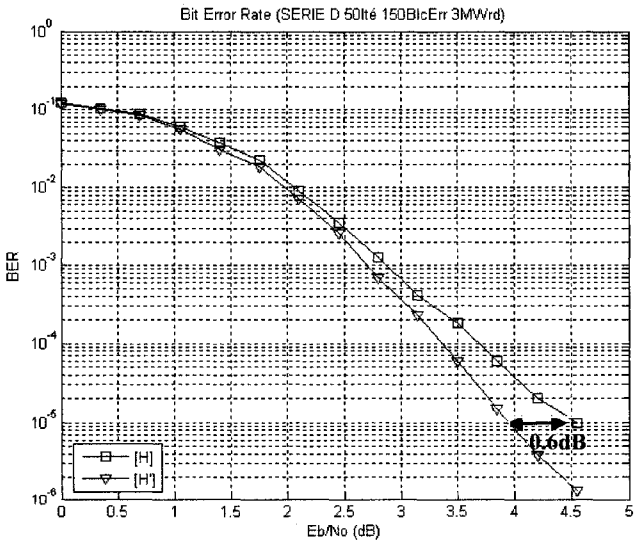


Figure 5.3: Performances d'erreur obtenues pour les matrices de parité **H** et **H'** (D13 et OpD13)[1]. Une amélioration de 0.6dB est obtenue à 10^{-5}.

[1] Des explications complémentaires sur les codes utilisés dans ce mémoire sont disponibles en Annexe IV.

5.3 Étude du cycle minimum moyen des codes LDPC

5.3.1 Origine

De nombreux articles discutent des cycles des codes LDPC [29] affirmant que le plus petit cycle d'un code est un critère déterminant. Ce critère est sensé, à matrices de mêmes dimensions, différencier les performances d'erreur.

Il est vrai que les cycles de petite longueur pénalisent fortement les performances des codes comme il a été mis en évidence au début de ce chapitre. Et ceci à tel point que certains liens du graphe de Tanner causent plus de dommage que de gain pour la correction des erreurs.

Cependant, pendant longtemps il nous était difficile de comprendre pourquoi certains codes avec beaucoup de cycles étaient plus performants que d'autres issus du même algorithme, avec les mêmes propriétés, mais avec moins de cycles. Un exemple de ceci est donné au tableau 5.1.

Code ID	4-Cycle	6-Cycle	8-Cycle	Perf. d'erreur
Z104 [1]	17	188	1253	-
Z101 [1]	22	163	1305	+

Tableau 5.1: Code à taux ½ de dimension 2000x1000 [3]

Cet exemple, mettant en relation la performance d'erreur avec le nombre de cycles, n'est pas du tout un cas isolé. Il est même fréquent pour les codes que nous avons examinés. Ceci laisserait penser que le nombre de cycles n'est pas aussi important que prévu. Seul le cycle minimum serait déterminant. Mais ce n'est pas ce qui est observé en réalité. Les parties suivantes tentent de répondre à ces interrogations.

[1] Des explications complémentaires sur les codes utilisés dans ce mémoire sont disponibles en Annexe IV.

5.3.2 Le cycle minimum moyen

Une explication au phénomène des cycles pourrait se trouver dans l'étude du cycle minimum (CM). Le schéma 5.4 illustre un graphe de Tanner où les cycles ont été mis en évidences.

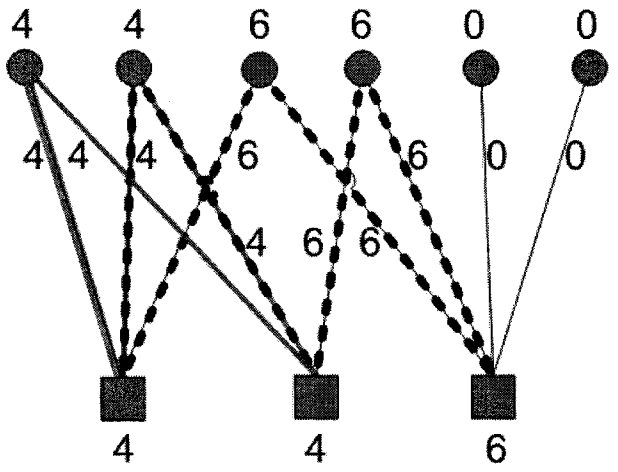


Figure 5.4: Graphe de Tanner avec les cycles minimaux des nœuds et des liens.

Ce code comporte un 4-cycle et 3 6-cycles. Le cycle de longueur 4 à été mis en évidence à gauche ( ) et un des cycles de longueur 6 à droite ( ). ▪▪▪▪▪

La présence du 4-cycle a pour conséquence de déterminer la longueur du cycle minimal du code à 4. Mais qu'en est-il du cycle minimum sur les nœuds et les liens ?

- Pour les nœuds du haut, les cycles minimums[1] sont {4, 4, 6, 6, 0, 0}
- Pour les nœuds du bas, les cycles minimums[1] sont {4, 4, 6}
- Pour les liens, les cycles minimums[1] sont {4, 4, 4, 4, 6, 6, 6, 6, 0, 0}

[1] Ces séquences reprennent les cycles minimaux qui ont été comptabilisés sur la Figure 5.4. Les chiffres sont donnés par lecture de gauche vers la droite sur cette même figure. Par la suite, cette suite de chiffres sera utilisée pour le calcul du cycle minimum moyen d'un code correcteur d'erreur.

Cette analyse permet d'obtenir plus de détails sur la répartition des cycles minimaux du graphe. Il est possible de connaître le CM de chaque nœud et de chaque lien du graphe de Tanner. Ces informations vont nous permettre d'affiner la valeur réelle du cycle minimum. Pour cela, nous allons sommer tout les CM de chaque lien (ou nœud) et diviser le résultat par le nombre d'adition réalisé plus 1. Ce calcul sera appelé par la suite CMM pour Cycle Minimum Moyen.

Le calcul du CMM peut être fait de trois façons différentes comme énoncé ci-après :

$$\textit{Avec les noeuds de variable:} \quad gnv = \frac{1}{n}\sum_{i=1}^{n} g_{vi} \tag{5.2}$$

$$\textit{Avec les noeuds de parité:} \quad gnc = \frac{1}{r}\sum_{i=1}^{r} g_{ci} \tag{5.3}$$

$$\textit{Avec les Liens (edges):} \quad ge = \frac{1}{E}\sum_{i=1}^{E} g_{ei} \tag{5.4}$$

avec g_{vi} le cycle minimum du nœud de variable i, g_{ci} le cycle minimum du nœud de parité i, g_{ei} le cycle minimum du lien i, et E le nombre de liens dans le graphe de Tanner.

En utilisant ces formules, nous obtenons les résultats suivants :

- Nœuds de variable: $gnv = \frac{4+4+6+6}{4} = 5$
- Nœuds de parité: $gnc = \frac{4+4+6}{3} = 4{,}7$
- Liens: $ge = \frac{4+4+4+4+6+6+6+6}{8} = 5$

Cette méthode est donc utilisée pour le calcul du CMM. Il est d'ailleurs observé par la suite, que pour des codes avec beaucoup de cycles, le CMM tend vers le cycle minimum du code.

5.3.3 Étude comparative du gnv, gnc et ge

Nous avons défini au cours de ce chapitre trois méthodes pour le calcul du cycle minimum moyen. Le *gnc* et *gnv* font un comptage en utilisant les nœuds de variable et de parité. Par contre, le *ge* utilise les liens du graphe de Tanner.

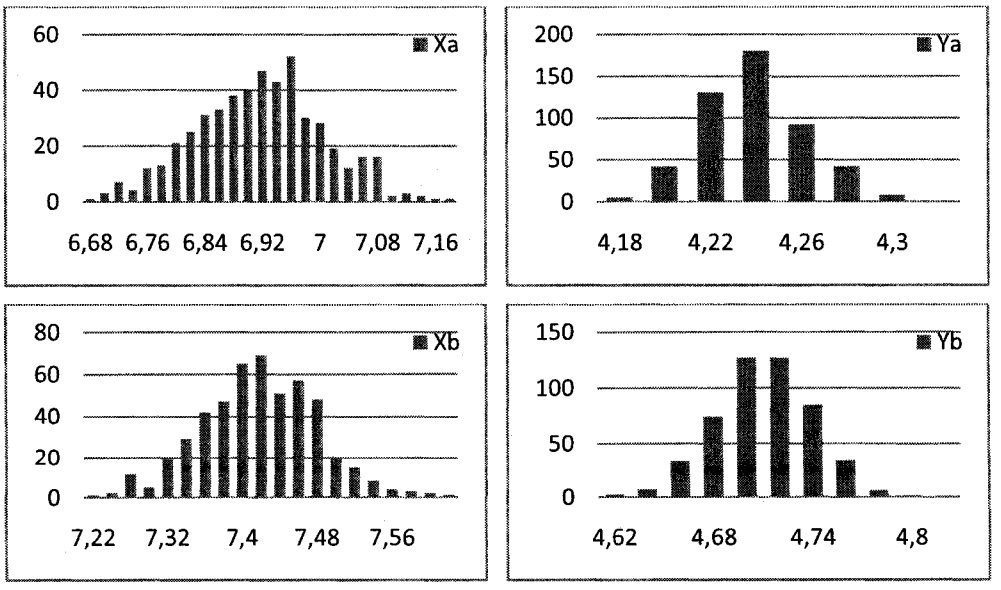
Essayons de savoir si ces CMM sont reliés entre eux. Pour cela, étudions des séries de 500 codes, dont les caractéristiques de construction sont résumées dans le tableau 5.2.

Nom des Série de Codes	Xa	Xb	Xc	Ya	Yb	Yc
n	500	1000	2000	500	1000	2000
r	250	500	1000	250	500	1000
O	3	3	3	6	6	6

Tableau 5.2: Liste des séries de codes utilisés pour l'étude.

avec O, le nombre de 1's par colonne dans **H**.

Les distributions de CMM sont obtenues ci-après. L'axe horizontal correspond au CMM et l'axe vertical au nombre de codes parmi les 500 étudiés, pour un CMM donné.



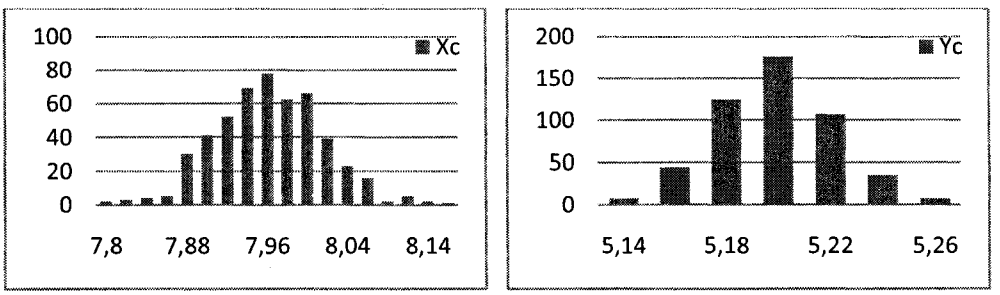


Figure 5.5: Distribution quasi-gaussienne du ge des codes des séries Xi, Yi

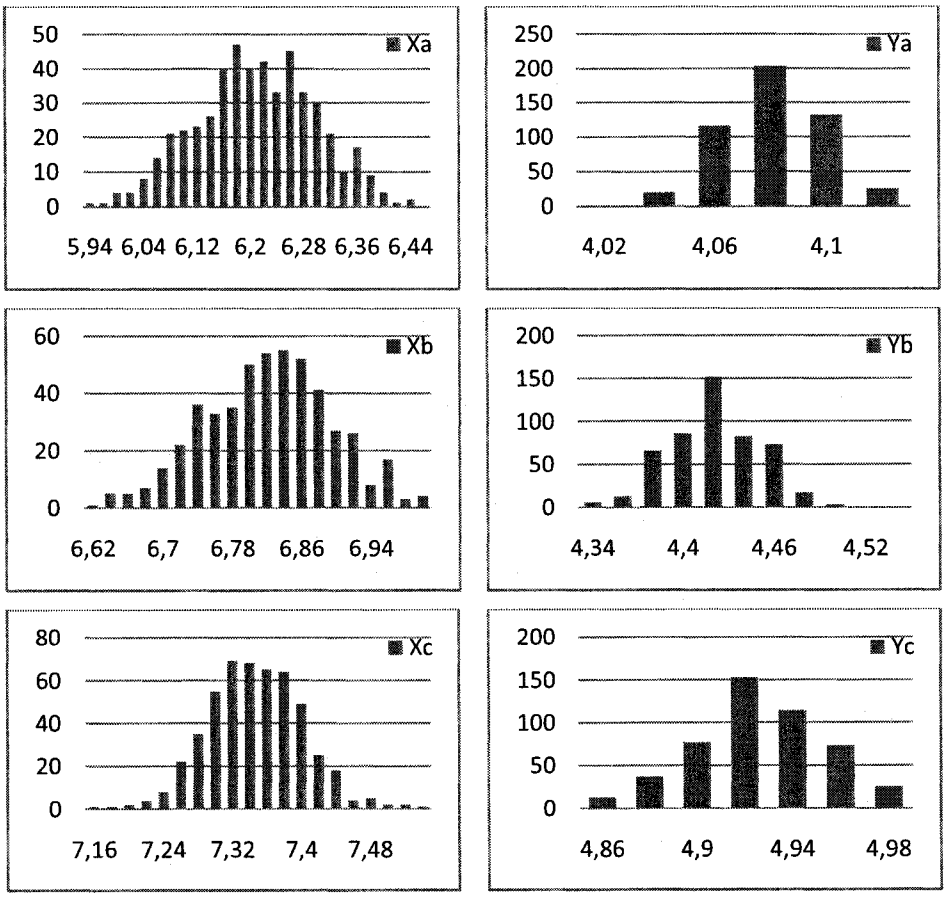


Figure 5.6: Distribution quasi-gaussienne du gnv des codes de la série Xi, Yi

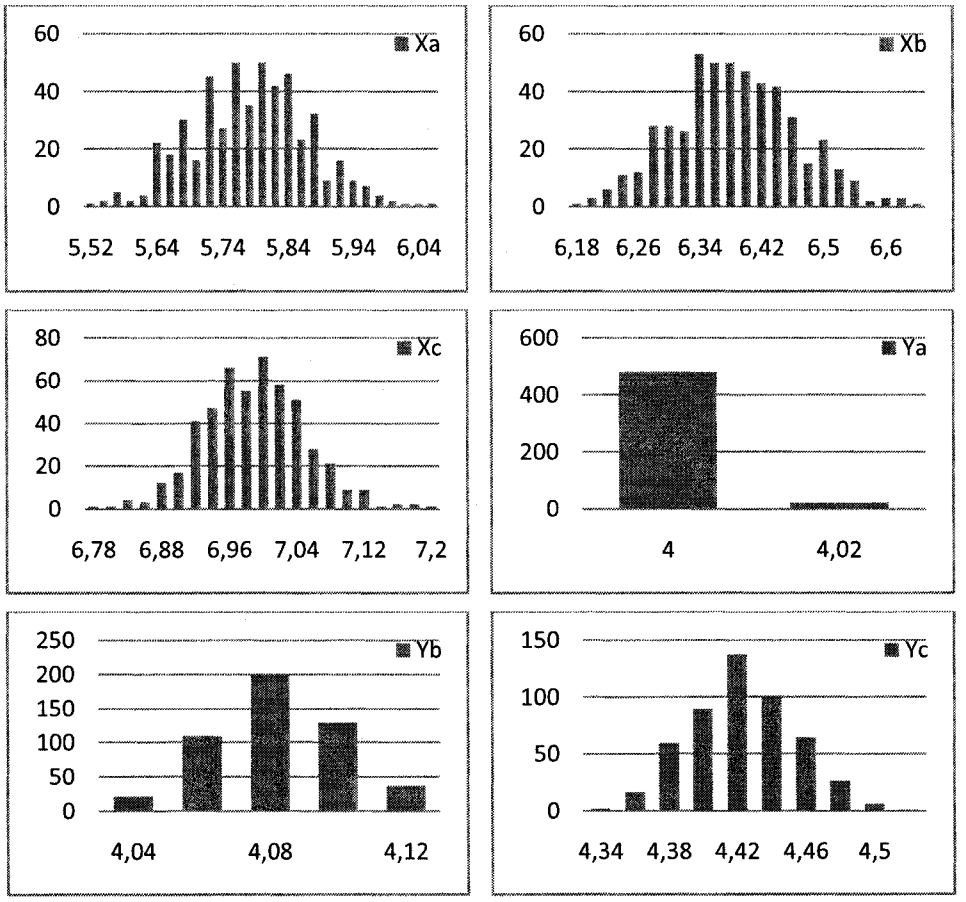


Figure 5.7: Distribution quasi-gaussienne du gnc des codes de la série Xi, Yi

Les figures précédentes montrent que le calcul du CMM n'apporte pas toujours d'information supplémentaire :

- Les matrices très creusent, ont une espérance (moyenne) élevée.
- Les matrices moins creuses peuvent avoir un CMM égal au cycle minimum. Ceci revient à la borne supérieure suivante :

$$g \leq g_x \quad avec\ g_x\ pouvant\ être\ remplacé\ par\ ge, gnv, gnc. \tag{5.5}$$

- Les distributions de gnv et gnc ont une variance réduite comparé à ge. Cela prouve que le ge est capable de mieux différencier les codes analysés.

Il est aussi intéressant d'étudier les valeurs de gnc et ge en fonction de gnv. Ceci va nous aider à convenir d'une seule et unique méthode pour le calcul du CMM. La Figure 5.8 présente ces fonctions. Pour obtenir ces graphes, nous avons trié par gnv croissant le tableau contenant les gnv, gnc, ge et 4-cycle des 500 codes.

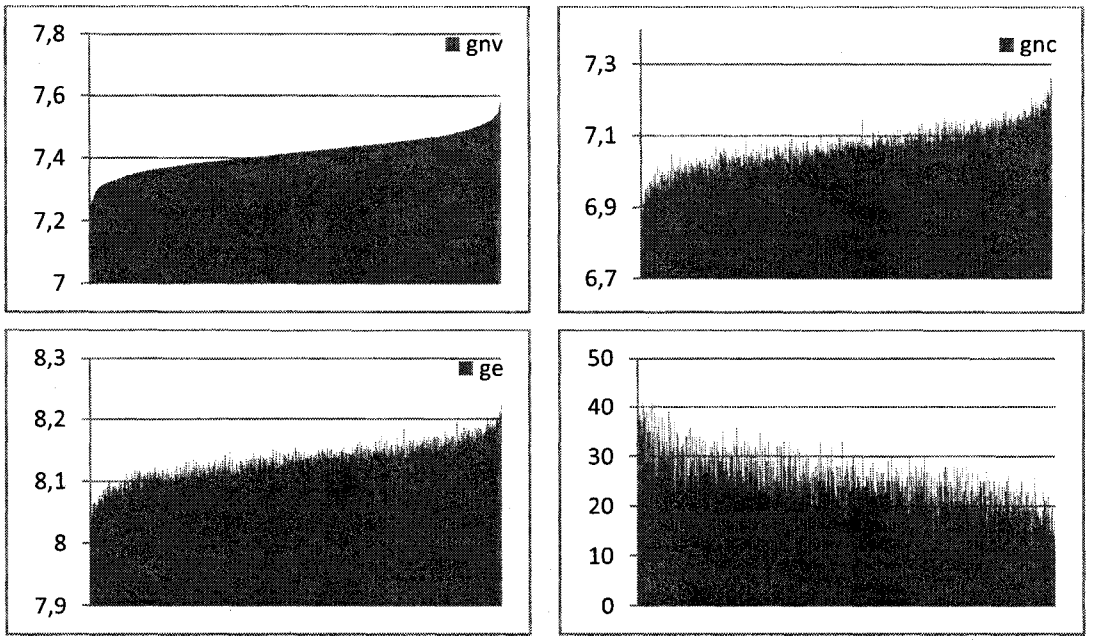


Figure 5.8: Évolution de gnc, ge et des 4-cycles en fonction de gnv[1]

Ces courbes montrent qu'il existe un lien fort entre les différentes méthodes de calcul du CMM. Ce qui prouve que les trois méthodes de calcul du CMM donnent quasiment les même résultats. Cependant, nous avons vu que l'espérance du ge est beaucoup plus élevée que les deux autres. Il semble donc que ce dernier soit plus adéquat pour le calcul du cycle minimum moyen lorsque le nombre de cycles minimaux est élevé. Il permet d'obtenir un CMM différent du cycle minimum (i.e. Figure 5.7.Ya) pour les matrices creuses et celles qui le sont moins.

[1] En raison du tri par ordre croissant de gnv, les quatre graphes n'ont pas d'échelle horizontale.

Au vu de ces résultats, seuls les ge et gnv seront utilisés dans la suite de ce mémoire pour le calcul du CMM. Les graphes de performance d'erreur utiliseront exclusivement le ge, qui donne toujours de meilleurs résultats que les deux autres.

5.3.4 Utilisation pratique du CMM

Pour vérifier l'existence d'une influence quelconque du cycle minimum moyen sur les performances d'erreur, trois approches vont être suivies. Elles permettront de cibler les cas où le calcul du CMM peut apporter des informations concrètes sur les performances d'un code.

1re Analyse: (Pour des codes de petites dimensions)

Avec l'algorithme de [3], générons 14 codes[1] de dimension (254, 128). Ces 14 codes sont simulés et analysés par les algorithmes de comptage de cycles. Le tableau 5.3 rapporte l'ensemble des résultats.

Code ID	A 29	A 71	A 100	A 79	A 87	A 1	A 17	A 131	A 51	A 95	A 12	A 147	A 139	A 42
4-Cycle	29	34	24	31	28	22	24	29	21	25	23	26	19	23
ge	5,49	5,40	5,63	5,46	5,60	5,64	5,58	5,51	5,64	5,57	5,57	5,55	5,72	5,63
gnv	5,21	5,14	5,42	5,21	5,35	5,42	5,32	5,20	5,43	5,35	5,32	5,29	5,51	5,39
Perf.	Mauvaise													Bonne

Tableau 5.3: Résultats de l'analyse des cycles

Le tableau 5.3 présente les *4-cycle*, ge et gnv en fonction des performances d'erreur. Les codes avec les moins bonnes performances d'erreur se trouvent à gauche du tableau tandis que les meilleurs codes sont à droite. Ce classement correspond au BER observé à 4.6 dB, à la Figure 5.9. Notez que la barre de dégradé sera utilisée par la suite pour classer les codes par performance d'erreur. Foncé si mauvaise et claire si bonne.

[1] La méthode pour retrouver les 14 matrices de parité des codes utilisés est expliquée en Annexe IV.

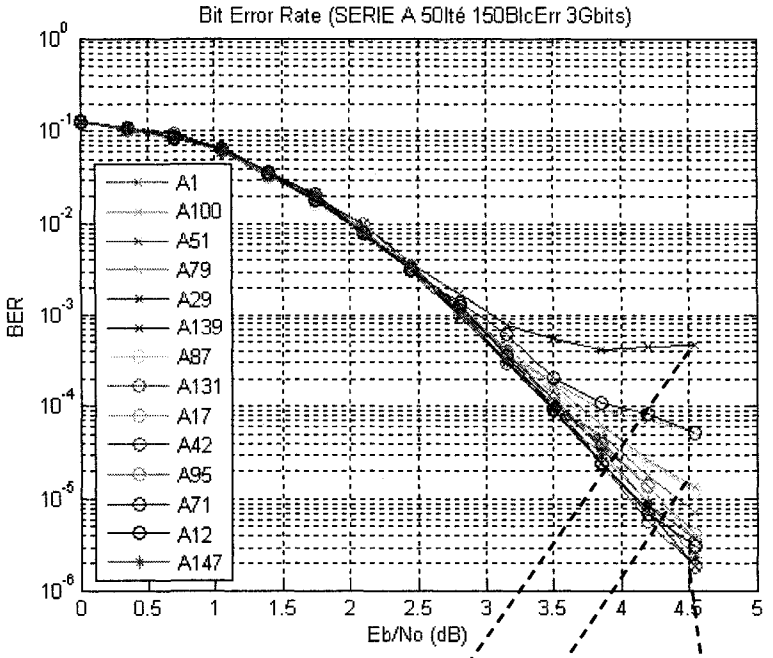


Figure 5.9: Performance d'erreur des 14 codes de la série A.

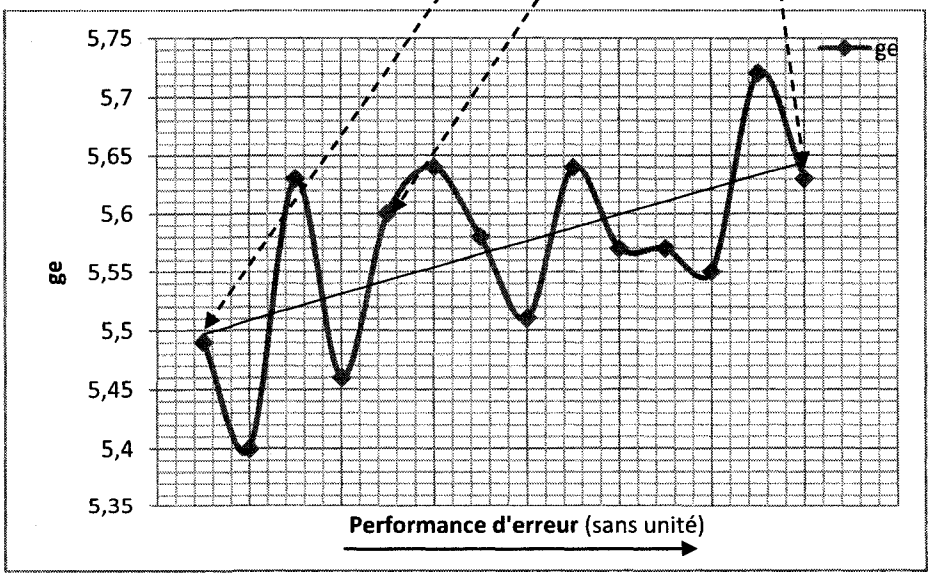


Figure 5.10: Évolution du *ge* en fonction de la performance d'erreur des codes

Ces premiers résultats sont intéressants. On observe ici une pente positive. La même observation à été faites lorsque nous avons renouvelé l'expérience. Cela prouve qu'il existe une relation entre le CMM et les performances d'erreur pour des petits codes. Cependant, l'utilité du cycle minimum moyen pour différencier les performances d'erreur de deux codes ayant les mêmes caractéristiques et provenant du même algorithme de construction ne fonctionne pas aussi bien que souhaité.

2ème analyse: (Pour des codes de moyenne dimension)

Cette seconde analyse cherche à généraliser le lien précédemment établi avec des codes de petite dimension.

Avec le même algorithme [3], générons 1500 codes de dimension (2200, 1100). Nous les nommons Ti , $i = 1 \ldots 1500$. Ces codes sont extrêmement longs à simuler et de ce fait un protocole de sélection est nécessaire.

L'idée est de démontrer qu'un code avec un cycle minimum moyen élevé devrait être meilleur dans la plupart des cas qu'un code avec un CMM faible. Le schéma 5.11 montre la distribution des ge pour les 1500 codes.

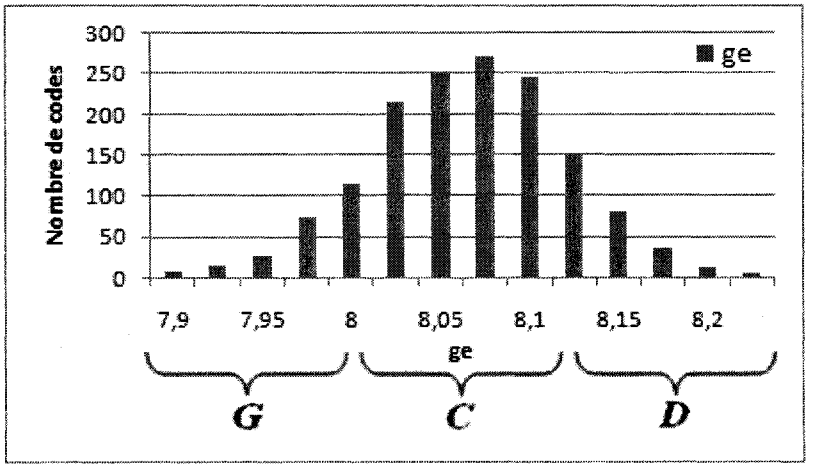


Figure 5.11: Distributions du ge pour les 1500 codes de la série T

Le protocole consiste à choisir des codes avec un ge élevé, puis de comparer leurs performances d'erreur avec celles d'autres codes de ge plus faibles. Le tableau 5.4 regroupe les résultats obtenus.

Code ID	T1187	T553	T349	T185	T58	T1071	T450	T356
4-Cycle	13	14	27	26	24	11	39	22
gnv	7,57	7,55	7,34	7,43	7,43	7,58	7,31	7,41
ge	8,21	8,21	7,99	8,08	8,06	8,20	7,96	8,05
Localisation	D	D	G	C	C	D	G	C
Perf. d'erreur	Bonne							Mauvaise

Tableau 5.4: Performance de certains codes de la série T

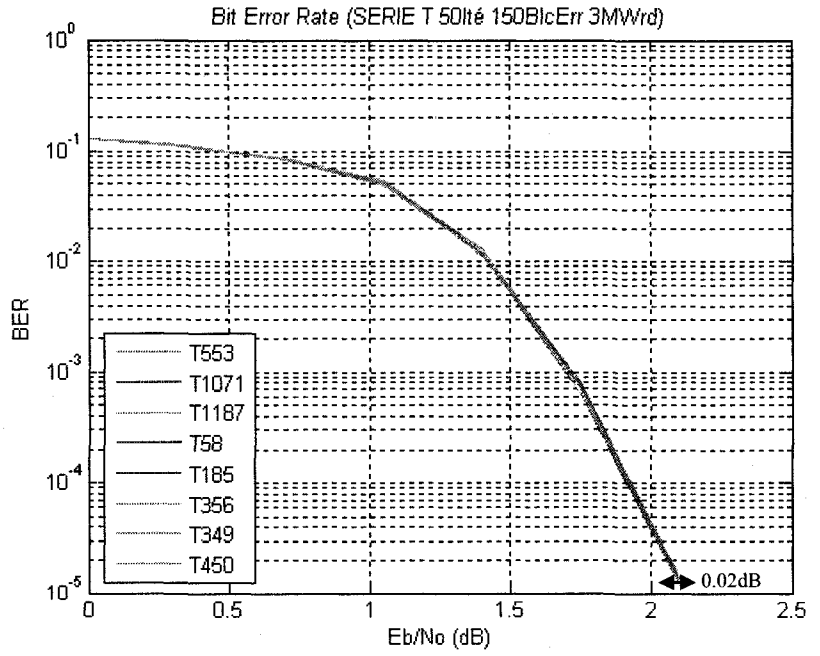


Figure 5.12: Courbes d'erreur de codes de la série T

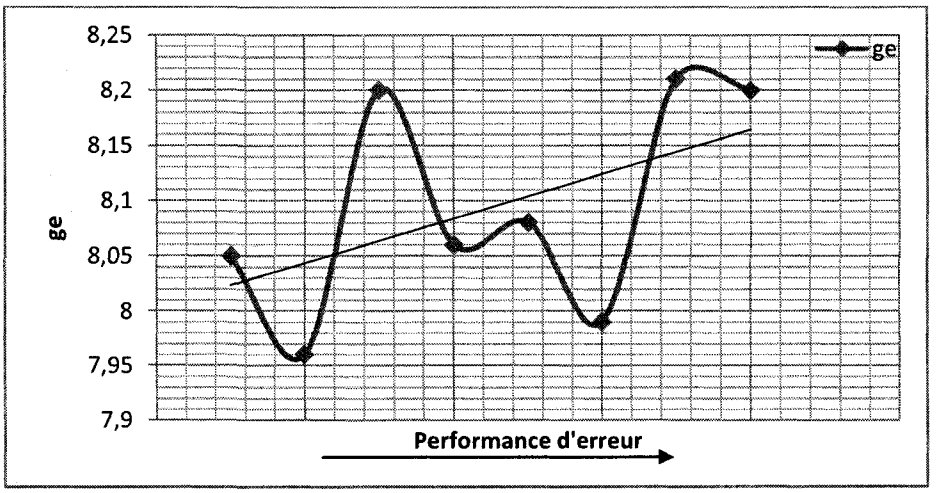


Figure 5.13: Évolution du *ge* avec la performance d'erreur des codes

Les résultats sont cette fois un peu plus mitigés. La même tendance d'augmentation est retrouvée, mais dans une proportion moindre. La courbe montre qu'un code se trouvant à droite (D) des courbes de la Figure 5.11 à plus de chance d'être performant qu'un autre se trouvant au centre (C) ou à gauche (G). Cependant, pour une performance d'erreur à 10^{-5}, il n'y a que 0.02dB de différence entre le meilleur et le plus mauvais des codes simulés. Il n'est pas possible dans ces conditions de demander à une méthode quelconque de trouver le meilleur code d'un groupe, sans simulation où la différence entre le meilleur et le moins bon est aussi faible. L'utilité du CMM pour des codes de dimension moyenne et plus grand semble limitée.

3ème Analyse: Variation du nombre de 1's dans les colonnes de **H**

Pour augmenter la différence de performance d'erreur entre les codes, nous allons faire varier un seul paramètre pour la génération des matrices. La différence de performances devrait augmenter et le CMM devrait pouvoir différentier les différents codes.

Pour cette expérience, nous avons généré deux codes par série (afin de confirmer les résultats obtenus), ayant comme numéro de séquence pseudo aléatoire 233 et 848. Les numéros de séquence ont été choisis arbitrairement. Veuillez vous référer à l'annexe IV pour plus de détails sur les numéros de séquence pseudo aléatoire.

Code série	Ha	Hb	Hc	Hd	He
4-Cycle moyen	0	5	28	121	346
ge moyen	∞	11,89	6,77	5,57	4,99
O	1	2	3	4	5
Perf. erreur	Mauvaise		Bonne		Mauvaise

Tableau 5.5: Caractéristiques et performances d'erreur des codes Hx.

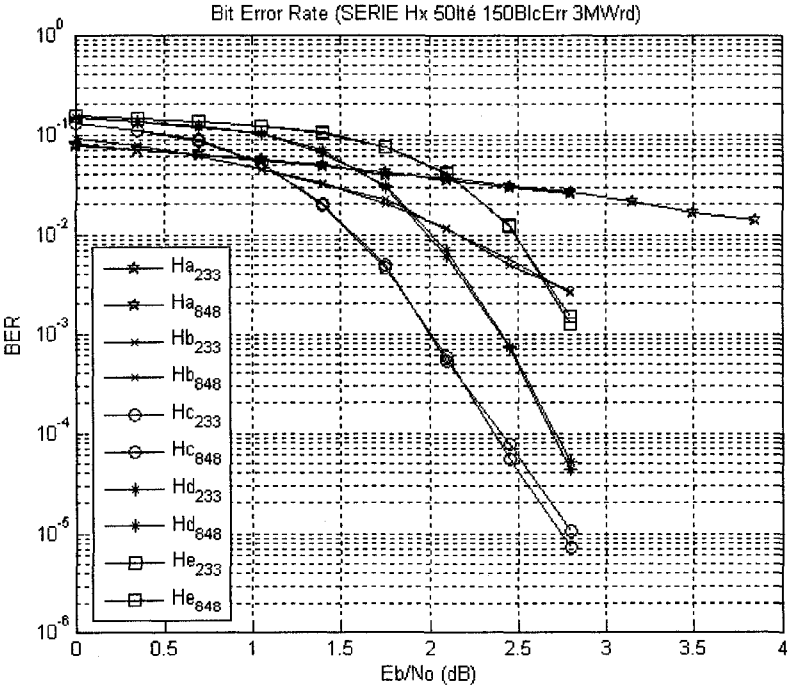


Figure 5.14: Courbes de performances d'erreur des codes de la série Hx.

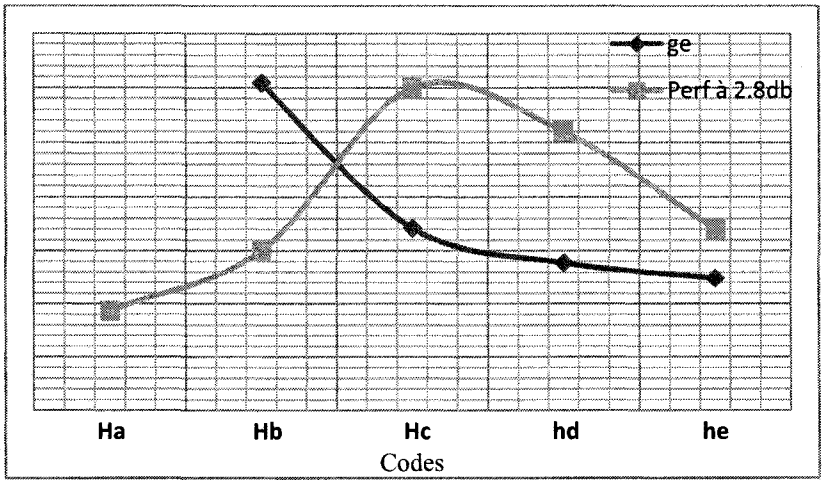


Figure 5.15: Évolution du ***ge*** et des performances d'erreur à 2.8dB .

Les courbes de la Figure 5.15 montrent clairement qu'il existe une plage de CMM pour laquelle les codes sont performants. On peut résumer les informations que nous apporte cette courbe de la manière suivante :

- Si le CMM s'approche du cycle minimum, les performances se dégradent nettement. Ceci explique que les codes avec un nombre élevé de 4-cycles sont peu performants.
- Si le CMM s'éloigne trop du cycle minimum, cela implique une matrice trop creuse, qui ne permet pas aux algorithmes de décodage de fonctionner correctement. Les performances d'erreur deviennent alors passables.

Note: Bien entendu, ces résultats sont au moins valables pour les codes LDPC réguliers de taux ½, contenant des 4-cycles. Nous invitons le lecteur à relire le premier paragraphe de la partie 3.4 qui explique théoriquement cette variation de performance à haut SNR.

5.4 Étude du nombre de cycles des LDPC par l'algorithme PER

Nous allons dans cette partie utiliser l'algorithme PER pour enlever des cycles de longueur supérieure à quatre. Ceci permettra de relativiser l'importance du cycle minimum d'un code correcteur d'erreur.

5.4.1 Augmentation du cycle minimum avec l'algorithme PER

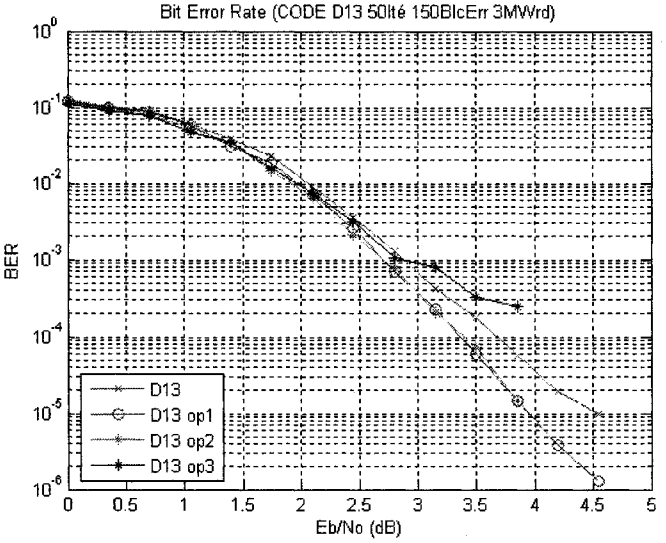
En 1963, Gallager [2] a déjà mentionné que les cycles de petites longueurs sont néfastes à l'obtention de bonnes performances d'erreur. Il a alors mis au point une méthode de construction évitant dans la mesure du possible, la formation de 4-cycles (cf Annexe I). Cependant, sa méthode de construction de code LDPC régulier ne permet pas de tous les éviter. C'est le cas pour l'ensemble des codes réguliers étudiés dans ce mémoire.
La méthode PER permet de montrer facilement que ces 4-cycles, en nombre très faible, sont néfastes pour les performances d'erreur, comme cela a déjà été mis en évidence à la Figure 5.3. La méthode montre que certains liens du graphe biparti font plus de mal que de bien aux performances d'erreur du code. Le tableau 5.6 résume ceci pour le code D13 précédemment étudié (partie 5.2).

	4-cycles supprimés	Liens supprimés	Amélioration à 10^{-5}	4-cycles	6-cycle	8-cycle	Liens restants
D13 optimisé	30	26	0.6 dB	0	138	913	742

Tableau 5.6: Bilan de l'optimisation du code D13 après suppression de tous les 4-cycles

Un code sans cycle de longueur 4 est obtenu après passage de l'algorithme de suppression des liens impliqués dans la formation de tel cycle. Il pourrait être intéressant de continuer cette suppression de lien pour obtenir un code sans cycle de longueur 6. Le code « D13 optimisé » sera par la suite noté « D13 op1 ».

Cependant, un problème majeur dans la méthode d'optimisation devient alors évident. La suppression de 138 6-cycles risque d'entraîner une disparition importante de lien dans le graphe de Tanner et donc une dégradation des performances d'erreur.
Afin d'éviter un tel comportement, la suppression d'une minorité des liens engagés dans la formation de 6-cycle est adoptée. Les optimisations op2 et op3 de la Figure 5.16 montrent les performances d'erreur lorsque 30 et 60 liens sont supprimés du graphe du code « D13 op1 ».



	4-cycle	6-cycle	# liens supp.
D13	30	197	0
D13op1	0	138	26
D13op2	0	79	56
D13op3	0	45	86

Figure 5.16: Performance d'erreur du code D13 après plusieurs optimisations successives. Un bilan des optimisations est présenté dans le tableau.

L'optimisation 2 a supprimé 43% des 6-cycles, mais l'absence de ces liens ne change rien aux performances. Il semble que ces derniers faisaient autant de bien que de mal. Leur déplacement dans une position où ils ne formeraient aucun 6-cycle aurait surement été plus souhaitable. Mais ceci n'est pas traité dans le présent mémoire.

Enfin, la Figure 5.17 montre les performances des codes Hc_{233} (code avec des 4-cycle), $Hc_{233\ Op1}$ (code avec des 6-cycle) et $Hc_{233\ Op2}$ (code avec des 8-cycle).

Le palier à haut SNR de $Hc_{233}op2$ est monté à cause de la diminution de la distance minimale. La suppression d'un nombre élevé de liens casse donc la structure même du code. Des comportements similaires sont observés sur l'ensemble des simulations où les cycles de longueurs 6 ont été retirés.

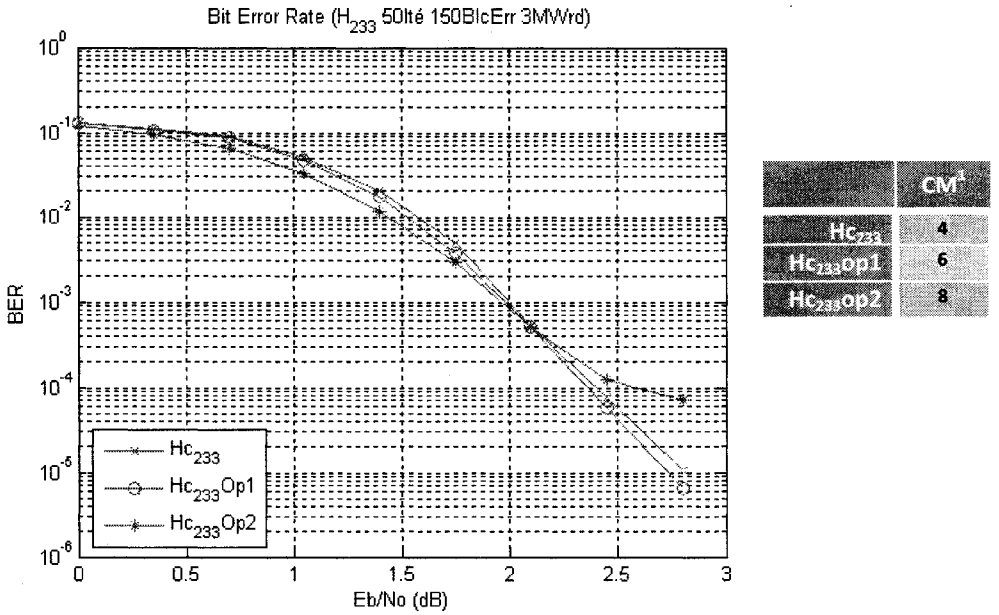


Figure 5.17: Performance d'erreur du code Hc_{233} en version 4-cycle ; 6-cycle et 8-cycle

5.4.2 Calcul théorique du nombre de cycles

La méthode de R. Neal [3] pour la construction de codes LDPC est certainement la plus répandue. L'algorithme peut construire des codes réguliers ou irréguliers en faisant varier les paramètres de construction. Ces paramètres influent essentiellement sur les degrés des nœuds ou sur la taille de la matrice. Cependant, le nombre de cycles est invariant à la taille de la matrice.

La formule 5.4 peut être utilisée pour le calcul du nombre de cycle moyen de longueur C = {4; 6; 8} pour les codes réguliers avec O [2] 1's par colonne.

---

[1] CM (Cycle Minimum) : Longueur du plus petit cycle présent dans un code.
[2] La lettre « O » désigne le nombre de 1's par colonne, dans une matrice de parité.

$$
\begin{aligned}
N_{CYC}(O,C) = {} & (2418C^2 - 24058C + 57521)O^4 \\
& -(2505C^2 - 23592C + 53975)O^3 \\
& +(12190C^2 - 1.10^{-6}C + 3.10^{-6})O^2 \\
& +(38117C^2 - 4.10^{-6}C + 9.10^{-6})O^1 \\
& +(24545C^2 - 2.10^{-6}C + 6.10^{-6})
\end{aligned} \tag{5.5}
$$

Cette formule construite de manière expérimentale donne une approximation sur le nombre de cycles contenu dans une matrice de parité d'un code de taux ½ construit avec la méthode de [3].

Par exemple, d'après la formule (5.5), pour une matrice de dimension 150 x 300, contenant trois 1's par colonne, le nombre de cycles de longueur 6 donne:

$$N_{CYC}(3,6) = 168 \text{ cycles.}$$

5.4.3 Le besoin de cycles

Dans la grande majorité des papiers publiés, les auteurs recherchent des codes sous forme d'arbre, qui n'abritent aucun cycle pour pouvoir utiliser le BP dans sa configuration optimale [30]. Il est vrai que des codes sans cycle n'auront jamais le retour au nœud d'origine, du message envoyé, à une itération ultérieure. Mais cette optimalité n'est vraiment pas souhaitable pour les cas pratiques, qui ne l'oublions pas, sont la finalité de toute recherche.

En effet, l'absence de cycles retarde les calculs de l'algorithme de décodage. Ceci a le lourd inconvénient d'augmenter la consommation d'énergie du décodeur, pour décoder l'information reçu. De plus, cela augmente inévitablement la latence de la communication.

Les communications en temps réel, qui représentent la quasi-totalité des systèmes de télécommunication, devraient rester l'objectif à atteindre. Le tableau 5.7 montre la latence et la consommation d'énergie observée sur trois codes LDPC de même taille, mais de cycles minimaux différents.

Code :	4-cycles	6-cycle	8-cycle
Latence (ms)	2.39	6.74	9.82
Consommation énergétique	24%	68%	100%

Tableau 5.7: Latence du codage et du décodage de codes 252x504 de cycles minimaux 4, 6 et 8.

Pour un système embarqué où l'énergie est peu disponible, même si il y a une différence de 0.6dB entre le code 4-cycle et le 8-cycle, ce dernier préférera probablement perdre de l'information pour multiplier par quatre l'autonomie de l'appareil.

5.5 Comparaisons des codes PEG et PER

La méthode de design utilisé pour ces deux types de codes est radicalement différente. Dans le cas des PEG [29], la matrice de parité est construite de manière progressive, en respectant certaines règles de construction. Cependant, avant d'ajouter un lien dans le graphe, l'algorithme vérifie que ce dernier ne formera pas de cycle de longueur inférieure à celle demandée à l'algorithme. Dans la majorité des cas, il est alors possible d'empêcher la formation de 4-cycles, 6-cycles ou plus.

La méthode d'optimisation développée dans ce mémoire optimise des codes existants par la suppression des liens participants à la formation de cycles. Il est possible de créer des codes pour n'importe quelle valeur de cycle minimum souhaité. Cependant, il est de plus en plus complexe de supprimer les cycles au fur et à mesure que l'on augmente la longueur du cycle minimum souhaité.

La méthode PEG est actuellement la plus utilisée pour la création de matrices de parité à cycles minimaux élevés (6, 8, 10...). Il est intéressant de comparer à latence identique les caractéristiques de ces deux classes de code LDPC et d'établir si la construction PER est intéressante pour le design de codes performants.

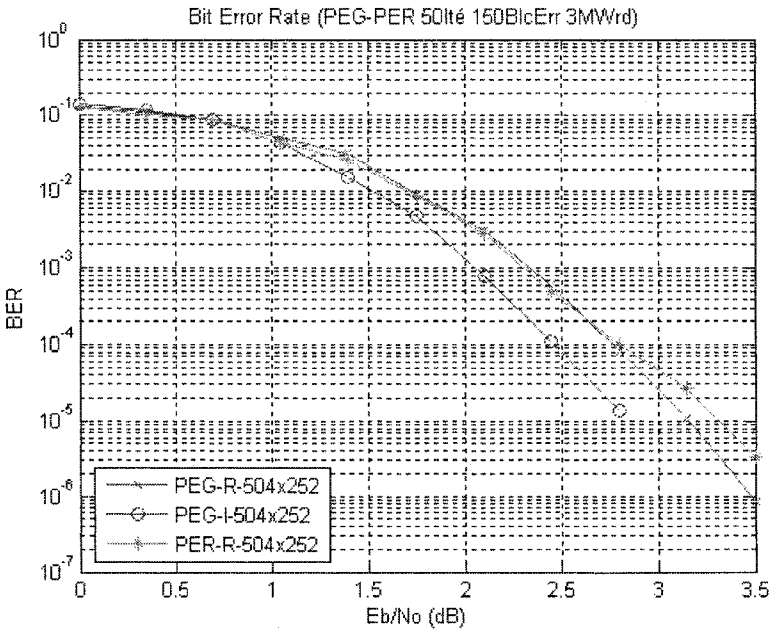


Figure 5.18: Performance des meilleurs codes PEG connu, de dimension 252x504 et du meilleur PER connu, de dimension 252x504. Le R / I indique respectivement la régularité ou l'irrégularité des degrés sur les nœuds.

La Figure 5.18 compare les performances du meilleur code 252x504 quasi régulier obtenu par la méthode PER, avec les meilleurs [31] PEG réguliers et irréguliers en date du 11/05/2005, de mêmes dimensions, sur un canal AWGN. Le code obtenu par la méthode décrite dans ce mémoire surpasse légèrement les performances d'erreur du PEG-régulier jusqu'à 2.8dB. Cela apporte un intérêt particulier à notre méthode d'optimisation de matrice de parité pour les faibles SNR. Après 2.8dB, le code PEG-régulier est meilleur.

Il est à noter que le code PEG-régulier comporte aussi des irrégularités légères au niveau des degrés sur certaines lignes.

Cependant, le code PEG-irrégulier surpasse nettement les performances des deux derniers codes, de plus de 0.35dB. Comme pour l'ensemble des codes irréguliers, ce

gain de performance à un prix. La Figure 5.19 présente la complexité moyenne de ces trois codes, qui à été obtenue en mesurant le temps moyen de codage et de décodage.

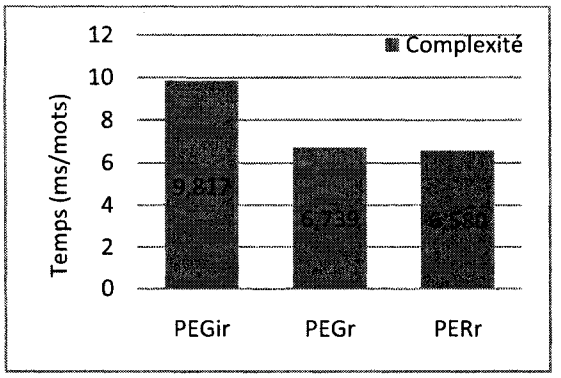


Figure 5.19: Temps de codage et de décodage d'un bloc,
pour 5 itérations au maximum de l'Algorithme Somme-Produit

Ainsi, le meilleur code PER est moins complexe pour le codeur et le décodeur que les deux autres. Son temps de codage-décodage pour un bloc n'est que de 6.58 ms, contre 6.74ms et 9.82ms pour les codes réguliers et irrégulier PEG.
En conclusion, notre méthode de construction de code LDPC peut être intéressante sur des canaux très bruités où l'énergie est peu disponible.

5.6 Conclusion

Ce chapitre a essayé de faire un bilan de l'influence des cycles sur les performances d'erreur des codes LDPC. Il a été vu que les performances d'erreur des petits codes sont souvent déviées, à haut SNR, de leurs courbes théoriques, ce qui a motivé particulièrement l'explication de ce phénomène. La mise en évidence de l'influence du cycle minimum moyen sur la déviation des performances vers la droite est une avancée importante à la compréhension difficile des cycles.

De même, la mise en évidence d'une valeur optimale du CMM pour des codes contenant des cycles de longueur 4 est un indice important pour la construction de codes irréguliers performants. Ce résultat est un indicateur pour déterminer le taux de remplissage adéquat d'une matrice de parité.

L'algorithme PER est une méthode originale pour la création de matrices de parité contenant des cycles de longueur 6. Il a été mis en évidence que cette méthode fonctionne systématiquement pour l'optimisation de matrice de parité construite par l'algorithme de Radford Neal [3].

Cependant, le gain de performance obtenu augmente de manière significative la complexité du codage, comme cela est déjà le cas avec les PEG. Une comparaison entre le meilleur PEG régulier et le meilleur PER quasi-régulier montre des performances semblables jusqu'à 2.8dB pour une complexité inférieure dans le cas du PER. Ceci donne un avantage certain au PER pour des systèmes à faible consommation énergétique dont le SNR est faible.

De plus, la mise en évidence pour les liens abritant des cycles de longueur 4, de leur contribution négative à la propagation des messages sur le graphe de Tanner est aussi une contribution importante à la compréhension de l'effet des petits cycles sur les performances d'erreur. Il a cependant été montré que les cycles de longueur 6, contribuaient davantage au bon passage des messages qu'a leur altération, même si cette dernière n'est plus à prouver.

Enfin, il semble que la volonté d'augmenter de manière excessive la valeur du cycle minimum n'est pas une solution souhaitable dans le cas d'implémentation réelle, puisque les besoins énergétiques de décodage augmentent considérablement [30]. Une focalisation sur des systèmes implémentables devrait être la priorité, plutôt que de faire la course à la capacité avec des codes de plusieurs millions de bits [32].

CHAPITRE 6

LES CYCLES ET LES CODES CSO2C

6.1 Introduction

Ce chapitre présente l'ensemble de nos résultats pour l'étude des cycles des codes convolutionnels doublement orthogonaux. Les mêmes outils que pour le chapitre précédent seront utilisés afin d'étudier l'importance des cycles pour les codes CSO2C.
Nous étudions tout d'abord les cycles sur des codes récursifs, et montrons les problèmes que cela implique. Dans une seconde partie, une étude comparative est réalisée sur les codes CSO2C-WS et SCSO2C-WS afin d'identifier leurs différences. Cette étude est basée sur le nombre de cycles de longueurs g et $g + 2$. Dans une troisième partie, nous utilisons l'algorithme de comptage des cycles sur les liens pour étudier différent codes et trouver une explication à la dégradation des performances des codes simplifiés[1]. Enfin, dans une dernière partie, nous utilisons l'algorithme de cycle minimum moyen sur des codes simplifiés de mêmes caractéristiques et établissons le lien entre CMM et performance d'erreur.

6.2 Mise à l'écart des codes récursifs

Nous nous proposons dans cette partie d'étudier les cycles d'un code récursif [2] au sens strict. Pour cela, nous devons au préalable déterminer sa réponse impulsionnelle, puis sa génératrice. Des problèmes de comptage seront mis en évidence, ce qui nous obligera à abandonner l'étude cyclique des codes récursifs.
Le code récursif convolutionnel doublement orthogonal au sens large suivant, de taux ½ sera pris en exemple dans la suite de cette partie.

---

[1] La définition des codes simplifiés sera présentée à la partie 6.3.2 de ce chapitre.
[2] Par récursif, nous désignons des codes CSO2C-WS classique avec pour seule différence, la présence d'une ou de plusieurs boucles de retour. La figure 6.1 illustre un codeur CSO2C-WS récursif (R-CSO2C-WS).

$$G_1: [0\ 1\ 32\ 103\ 462\ 1083] \tag{6.1}$$

$$G_2: [5\ 17\ 219\ 731] \tag{6.2}$$

Ces deux générateurs listent les connexions entre le registre à décalage et les sommateurs modulo-2, comme présenté à l a Figure 6.1.

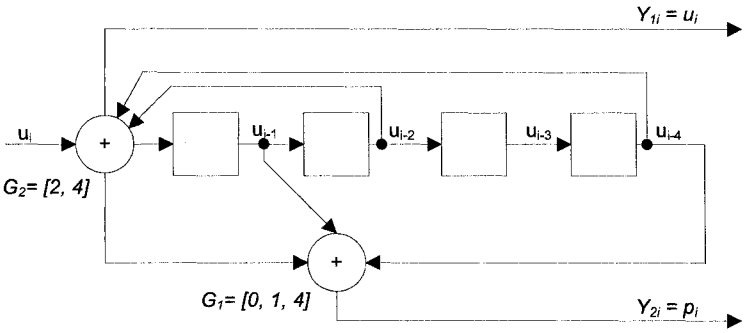


Figure 6.1: Schéma de principe d'un codeur R-CSO2C-WS
où les générateurs $G1 = [0, 1, 4]$ et $G2 = [2, 4]$ sont pris en exemple

Le générateur (6.1) est utilisé pour les connexions avec le premier sommateur modulo-2. Le générateur (6.2) est quant a lui utilisé avec le deuxième sommateur modulo-2.
Comme il a été défini au chapitre 2, le calcul de la réponse impulsionnelle de ce codeur est nécessaire à l'obtention de sa matrice génératrice, qui est à son tour utilisée par les algorithmes de comptage de cycles.
Ce codeur a été modélisé sous Matlab. Le début de sa sortie, à la réponse impulsionnelle d'entrée est donné ci-après :

$$Entrée = 1000000000000000000000000000000000000000000000000 \ldots \tag{6.3}$$

$$Sortie = 1101000000010100000001010000000101010100010100 0 \ldots \tag{6.4}$$

Un nombre important de 1's dans la séquence transmise sur le canal est observé. Ce qui indique inévitablement la présence d'un nombre important de petits cycles.

Il est maintenant possible de construire la matrice génératrice de ce code. La connexion la plus éloignée dans (6.1) est $\gamma_{max} = 1083$. La taille de la matrice est alors donnée par la relation suivante :

$$n = \frac{\gamma_{max} + 1}{R} \tag{6.5}$$

$$r = \gamma_{max} + 1 \tag{6.6}$$

avec R, le taux de codage.

Nous obtenons alors la génératrice avec la méthode de construction du chapitre 2. Cette matrice est analysée par l'algorithme de comptage de cycles. Le tableau 6.1 donne les résultats du calcul.

	r x n	4-Cycles	Temps calcul	Mémoire
Code R	1084 x 2168	3 697 229 219	22sec	11,1 Mo

Tableau 6.1: Comptage des cycles du code récursif R-CSO2C-WS (6.1), (6.2)

Ces résultats sont très intéressants pour montrer les performances de comptage de l'algorithme. Même un nombre élevé de connexions dans le graphe de Tanner ne réussit pas à le ralentir. Ceci n'est pas vrai avec le premier algorithme de comptage que nous avions présenté, qui « se fait fermer » par Windows après une consommation de plus 2Go de mémoire vive.

Cependant, ces résultats sont beaucoup moins intéressants pour une interprétation des cycles. En effet, avec un nombre aussi élevé de cycles, aucune interprétation ne peut être réalisée par l'étude des cycles sur cette famille de code. C'est pourquoi, dans la suite de ce mémoire, l'étude des codes récursifs est abandonnée.

6.3 Analyse cyclique des codes CSO2C

Nous allons présenter dans cette section l'analyse des cycles faite sur les codes CSO2C-WS et S-CSO2C-WS [4]. À la suite de cette étude, le comportement des cycles aura été défini et il deviendra possible de calculer empiriquement le nombre de cycles approximatifs, de longueur six et huit, pour ces deux classes de codes.

6.3.1 Les codes au sens large

Les codes convolutionnels doublement orthogonaux au sens large garantissent l'indépendance des observables jusqu'à la deuxième itération de l'algorithme de décodage sans recouvrir à l'usage d'entrelaceurs. Des conditions particulières sur les connexions entre les sommateurs modulo-2 et le registre à décalage doivent être validées pour obtenir une telle indépendance, ce qui revient à l'élimination des cycles de longueur 4.

Définition 6.1: [5]

Un code convolutionnel systématique de taux R= ½ est doublement orthogonal si et seulement si les positions γ_j, $j = \{1, 2, \dots, J\}$ *satisfont les propriétés suivantes :*

- *les différences simples* $\gamma_j - \gamma_k$ *sont distinctes*
- *les différences doubles* $(\gamma_j - \gamma_k) - (\gamma_m - \gamma_l)$ *sont distinctes des différences simples.*
- *Les différences doubles* $(\gamma_j - \gamma_k) - (\gamma_m - \gamma_l)$ *sont distinctes entre elles.*

où $(j, k, l, m) \in \{1, 2, \dots, J\}$ *tels que* $j \neq k,\ l \neq m,\ k \neq l$ *et* $m \neq j$.

Cependant, cette définition ne permet pas toujours d'obtenir un ensemble d'observables parfaitement décorélé. La permutation des indices k et m produit des différences doubles $(\gamma_j - \gamma_k) - (\gamma_m - \gamma_l)$ identiques de façons inévitables.

Ces répétitions inévitables ne peuvent être éliminées du processus de décodage sans générer une propagation des erreurs d'une itération à une autre, à partir de la 2ème itération. Ceci se traduit par la présence de cycles de longueur six.

Le tableau ci-dessous fourni le CMM et les cycles de longueur six et huit pour les codes CSO2C-WS les plus court, trouvé par [4].

J	Connexions $\mathcal{A} = \{\alpha_0, \alpha_1, \ldots \alpha_{J-1}\}$												6-cyc	8-cyc	ge
5	0	1	24	37	41	-	-	-	-	-	-		21	8	8,1477
6	0	1	17	70	95	100	-	-	-	-	-		147	136	6,283
7	0	1	53	128	207	216	222	-	-	-	-		369	452	6,2759
8	0	43	139	322	422	430	441	459	-	-	-		426	190	7,4862
9	0	9	21	395	584	767	871	899	912	-	-		3827	9846	6,1182
10	0	29	40	43	1020	1328	1495	1606	1696	1698	-		14040	49950	6,1714
11	0	220	521	695	908	926	1059	2457	3367	3458	3490		85796	492162	6, 0153
12	0	48	212	1014	1381	2217	4198	4373	4766	4885	4914	5173	66699	291992	6,0399

Tableau 6.2: CMM des liens (ge) et cycles de longueur six et huit pour des codes CSO2C-WS

Une étude de ces cycles en fonction du nombre de connexions J peut-être intéressant pour mesurer l'indépendance des observables. Les courbes suivantes évaluent le nombre de cycles de longueur six et huit ainsi que le rapport des cycles huit sur six et le cycle minimum moyen des liens en fonction de J.

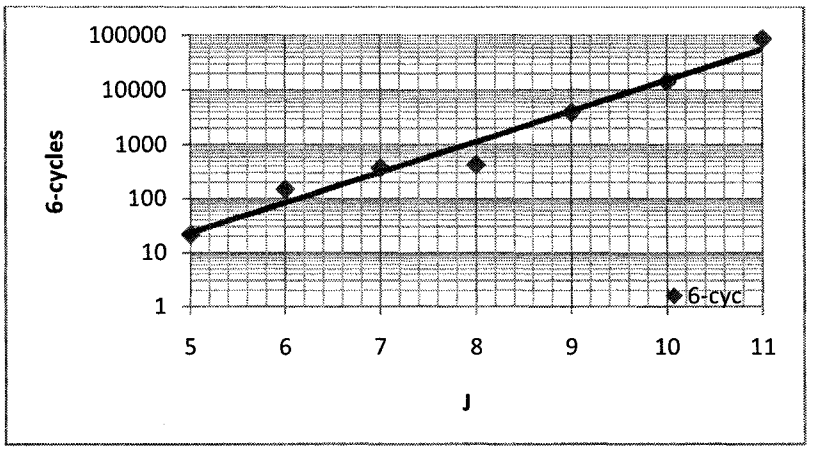


Figure 6.2: Nombre de 6-cycle en fonction de J pour les CSO2C-WS connus

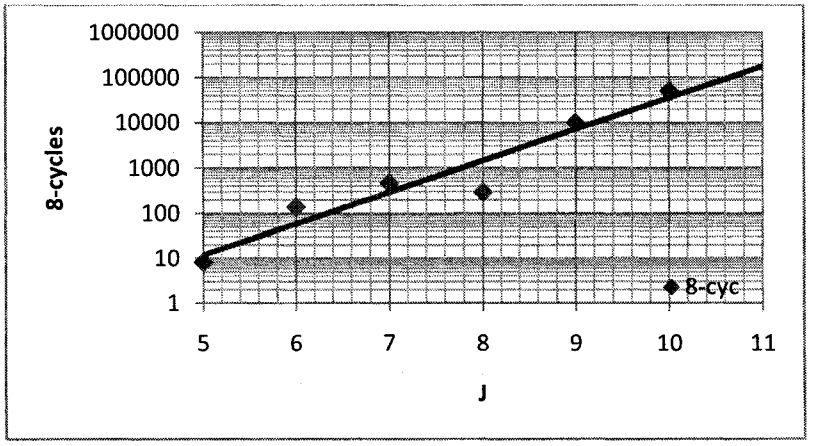


Figure 6.3: Nombre de 8-cycle en fonction de J pour les CSO2C-WS connus

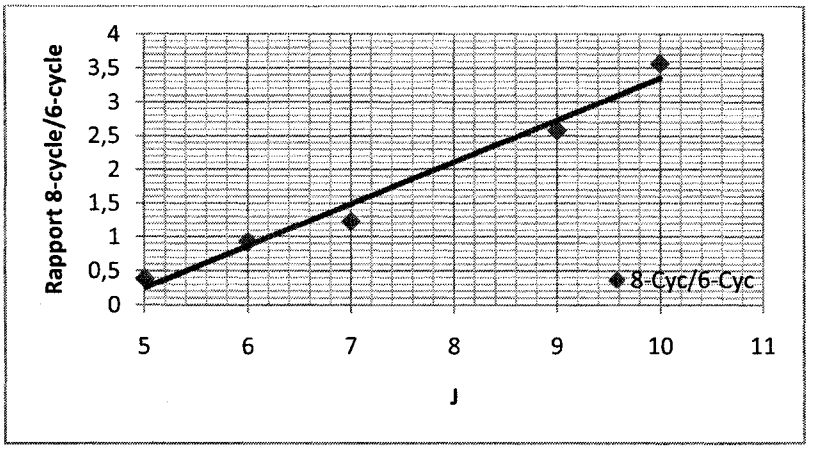


Figure 6.4: Rapport 8-cycle / 6-cycle en fonction de J pour les CSO2C-WS connus

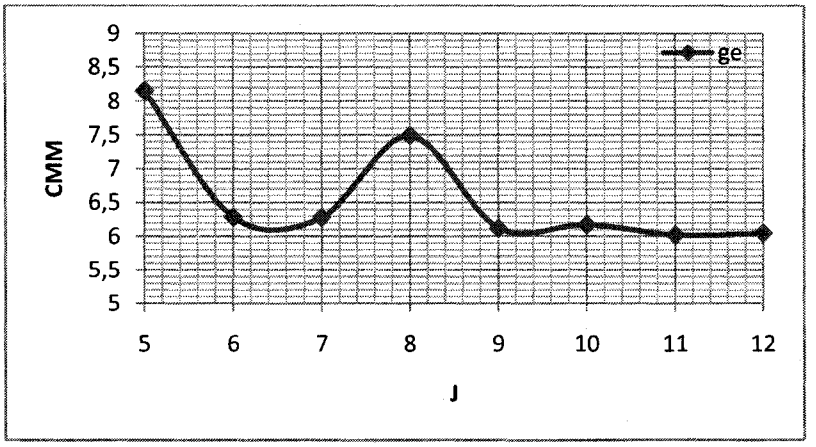


Figure 6.5: Cycle minimum moyen des liens en fonction de J
pour les CSO2C-WS connus

Ces courbes montrent un certain nombre de tendances, à savoir :

- Le nombre de cycles de longueurs six et huit augmente exponentiellement avec le nombre de connexions sur le registre à décalage J (Formules (6.7) et (6.8)) .
- Le rapport entre les 8-cycles et les 6-cycles est linéaire avec J.

- Le CMM à tendance à être très élevé pour les petits codes et à tendre vers le cycle minimum pour les plus grands.
- Le code pour $J = 8$ à un comportement particulier par rapport aux autres codes de J différent et ses valeurs doivent être ignorées.

Le nombre approximatif de cycles de longueurs six et huit, pour les codes CSO2C-WS est donné par les formules empiriques suivantes:

$$NbrCyc_6 = 0{,}034.exp^{1{,}3\,J} \tag{6.7}$$

$$NbrCyc_8 = y = 0{,}003.exp^{1{,}6\,J} \tag{6.8}$$

$$NbrCyc_8 = (0{,}620.J - 2{,}857).NbrCyc_6 \tag{6.9}$$

6.3.2 Les codes simplifiés

Les codes convolutionnel doublement orthogonaux simplifiés au sens large (S-CSO2C-WS) [4] représentent une classe de codes pour lesquels la condition exigeant que les différences doubles soit unique, à été relaxé. Nous définissons ces codes ci-dessous.

Définition 6.2: [5]

Un code convolutionnel systématique de taux R= ½ est doublement orthogonal et simplifié si et seulement si les positions γ_j, $j = \{1, 2, \ldots, J\}$ *satisfont les propriétés suivantes :*

- *les différences simples* $\gamma_j - \gamma_k$ *sont distinctes* $\forall\ k \neq j$;
- *les différences doubles* $(\gamma_j - \gamma_k) - (\gamma_m - \gamma_l)$ *peuvent être identiques* $\forall\ j \neq k,\ l \neq m,\ k \neq l\ et\ m \neq j$; *et* δ *est utilisé pour mesurer le pourcentage de ces différences identiques* [4];
- *Les ensembles des différences simples et doubles sont disjoints,* $S \cap D = \emptyset$.

Les codes simplifiés sont caractérisés par le facteur δ [4], qui indique le degré de non conformité d'un code par rapport à la définition 6.1, associée aux codes CSO2C-WS. Ce facteur représente le rapport entre les différences doubles égales positives N_d^e et le nombre total de différences doubles positives N_d. Le rapport de simplification s'écrit donc :

$$\delta = \frac{N_d^e}{N_d} \quad , 0 \leq \delta < 1 \tag{6.10}$$

Un ensemble important de codes simplifiés [4] ont été étudiés. L'Annexe III regroupe les résultats numériques des algorithmes pour cette famille de code.

Comme pour les codes au sens large, une analyse des cycles des codes simplifiés est faite ci-après.

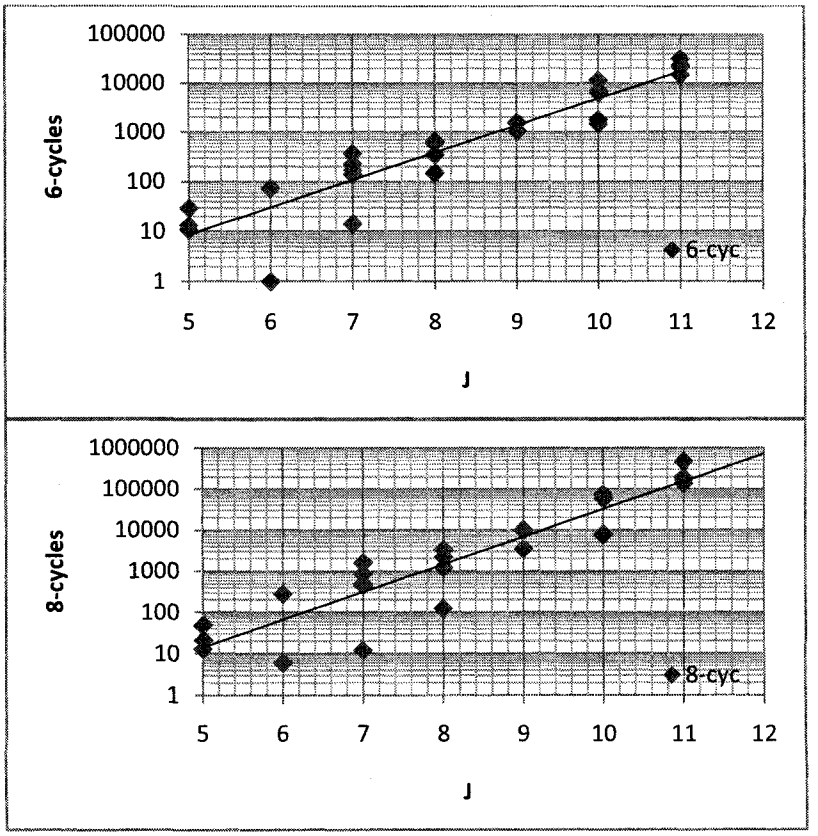


Figure 6.6: Cycles de longueurs 6 et 8 en fonction de J pour les S-CSO2C-WS

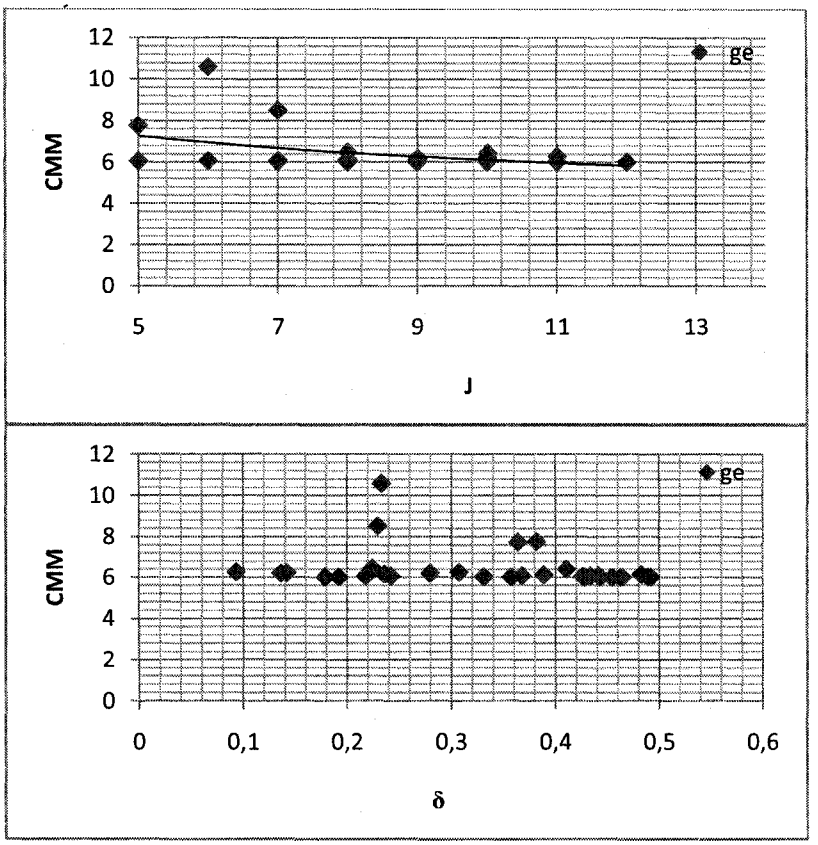


Figure 6.7: CMM des liens en fonction de J et δ pour les S-CSO2C-WS

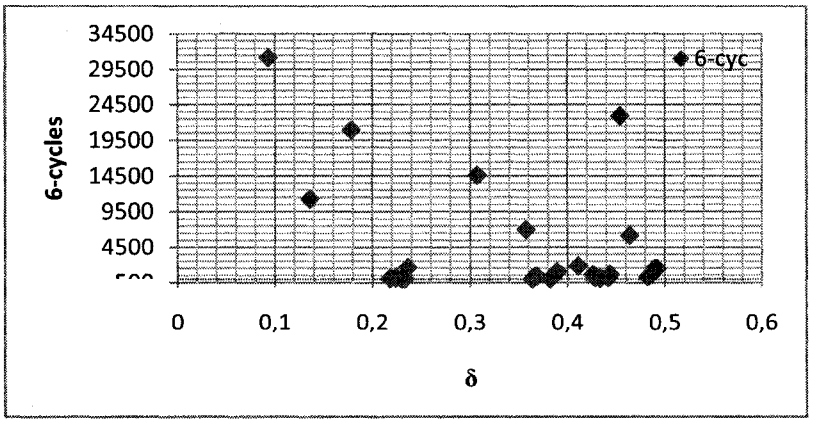


Figure 6.8: Cycles de longueurs 6 en fonction de δ pour les S-CSO2C-WS

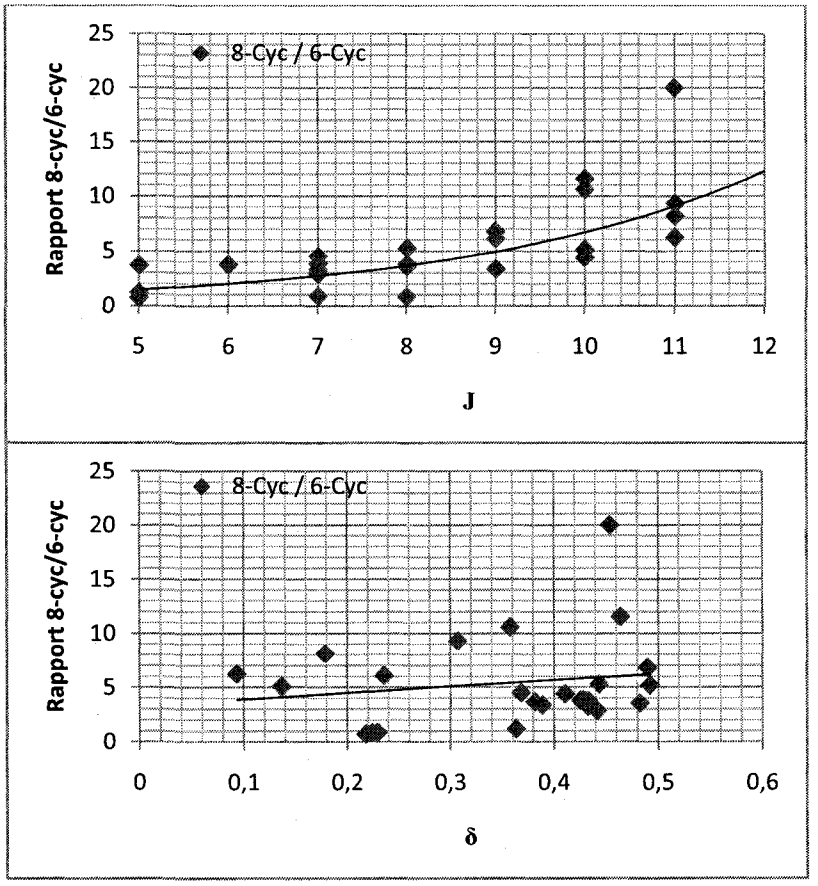


Figure 6.9: Rapport 8-cycle / 6-cycle en fonction de J et δ pour les S-CSO2C-WS

Ces courbes montrent un certain nombre de tendances, à savoir :

- Le nombre de cycles de longueurs 6 et 8 augmente exponentiellement avec le nombre de connexions sur le registre à décalage J (Formules (6.11) et (6.12)).
- Le rapport entre les 8-cycles et les 6-cycles augmente exponentiellement avec J. Ceci n'a pas été observé chez les codes au sens large. La relaxation des simplifiés semble augmenter le nombre de 8-cycles par rapport aux 6-cycles.
- Le CMM à tendance à être élevé pour les petits codes et à tendre vers le cycle minimum pour les plus grands. Il est indépendant de δ.

- Le rapport entre les 8-cycles et les 6-cycles augmente linéairement avec δ. La relaxation des simplifiés semble avoir des conséquences sur les cycles. Une étude plus poussée de ce phénomène est présentée dans la prochaine section.
- Le nombre de 6-cycles et 8-cycles ont tendance à être plus faible pour les codes simplifiés.

Le nombre approximatif de cycles de longueurs 6 et 8, pour les codes SCSO2C-WS est donné par les formules empiriques suivantes :

$$NbrCyc_6 = 0{,}015.exp^{1{,}268.J} \tag{6.11}$$

$$NbrCyc_8 = 0{,}006.exp^{1{,}546.J} \tag{6.12}$$

$$NbrCyc_8 = (0{,}321.exp^{0{,}303.J}).NbrCyc_6 \tag{6.13}$$

En conclusion, des propriétés identiques ont été trouvées chez les deux classes de codes. Cependant, le rapport entre les 8-cycles et les 6-cycles ne varie plus de la même façon et pourrait expliquer la dégradation de performance des codes simplifiés. La section suivante est dédiée à l'étude de ce phénomène.

6.4 Recherche d'irrégularité dans les codes doublement orthogonaux

Au cours de ce mémoire, nous avons mainte fois discuté des pertes de performances liées à la présence de petits cycles. Il est maintenant indéniable que leur étude est nécessaire pour comprendre les raisons qui causent les mauvaises performances d'un code.

Nous proposons dans cette partie de visualiser ces cycles sur les liens du graphe de décodage. Ceci permet de repérer des irrégularités de construction pouvant expliquer la dégradation de performance, même si aucune preuve ou démonstration ne conforte cette idée ou la réfute.

L'algorithme utilisé pour réaliser ces images est celui du chapitre 4 qui compte le nombre de cycles minimaux passant par un lien du graphe de Tanner. Pour cela, il construit la matrice génératrice du code, obtient le graphe de Tanner associé à cette matrice, puis le parcourt à la recherche de cycles. Il compte ainsi le nombre de cycles de longueur minimale passant dans chaque lien.

Dans le graphe de Tanner, les liens relient un nœud U_i, $i = 1, 2, \ldots, n$ et un nœud V_j, $j = 1,2, \ldots, r$. L'algorithme enregistre alors le nombre de cycles par liens aux coordonnées (i, j) dans une matrice de taille (r, n), identique à la matrice génératrice.

Les figures suivantes présentent l'image de codes CSO2C au sens large et simplifiés avec $J = 10$ et essaye de trouver des irrégularités pour expliquer la dégradation de performance des simplifiés.

Afin d'augmenter le contraste de la première matrice de la Figure 6.10(a), l'image utilise un regroupement des points quatre par quatre. Sans ce regroupement, la couleur de l'image serait uniforme et aucunes caractéristiques ne seraient visibles.

Les couleurs claires indiquent un nombre élevé de cycles et inversement pour les sombres. Une échelle du nombre de cycles se trouve à la droite de chaque image.

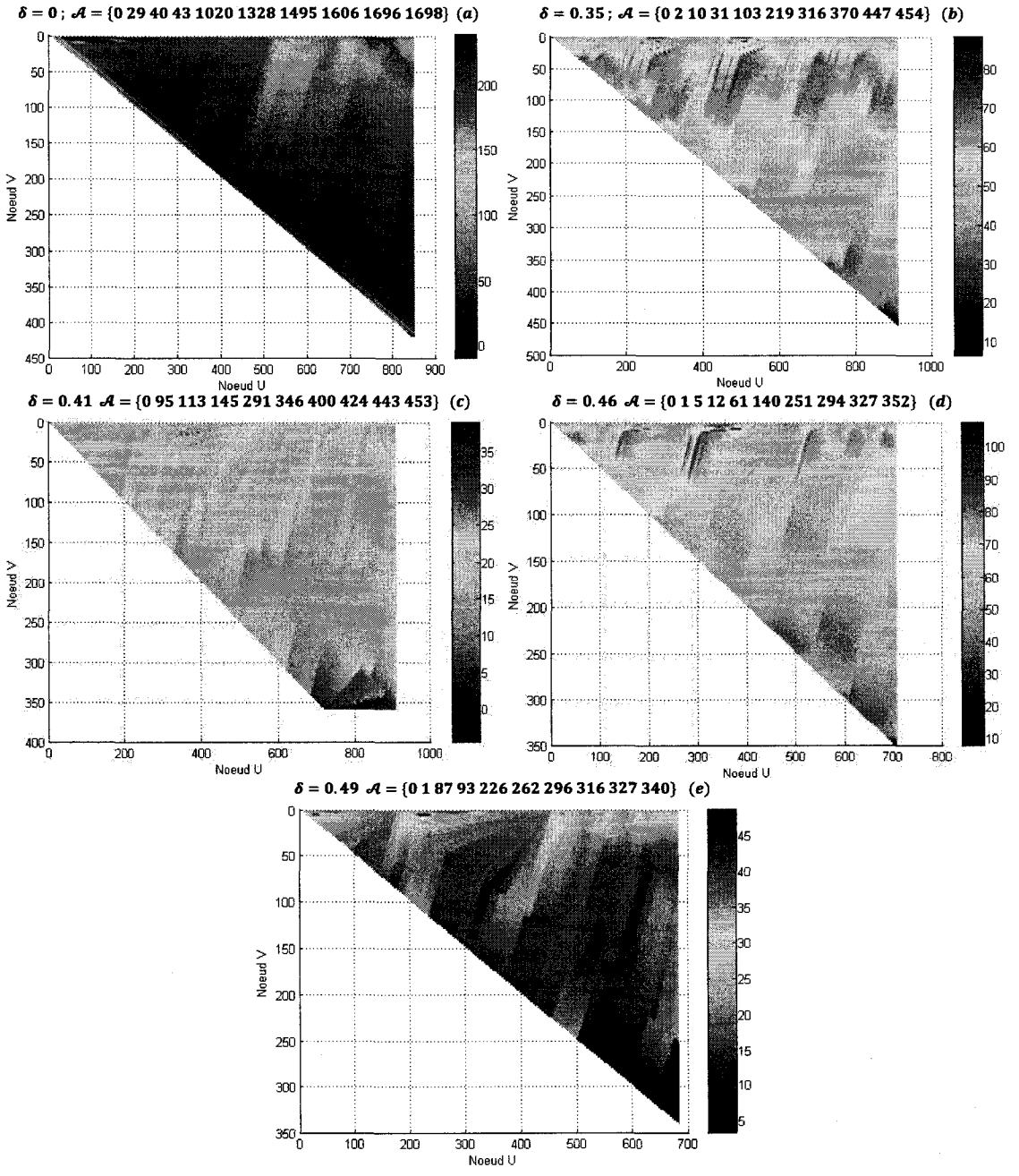


Figure 6.10: Distributions des cycles sur les liens du graphe de Tanner de code CSO2C-WS et S-CSO2C-WS avec $J = 10$. Les couleurs traduisent le nombre de 6-cycles sur les liens.

La Figure 6.10 présente les résultats de la distribution des cycles sur les liens du graphe de Tanner. Le code non simplifié (a) montre une concentration des cycles sur la diagonale et sur la partie haute du triangle ainsi qu'un nombre faible et relativement constant sur le reste de la matrice. Ces observations sont systématiques pour les codes avec $J > 7$.

Les codes simplifiés sont classés selon les valeurs du rapport δ. On remarque que pour le (c), le bas de l'image est absent. Cela traduit une absence totale de 6-cycles dans la région des U_i élevés sur le graphe de Tanner, causé par la diminution du nombre de liens en bout de matrice.

De plus en plus d'irrégularités sont observées sur les images. Pour certains codes (d, e), il est même possible de compter le nombre de connexions au registre à décalage.

Il apparait que relaxer certaines différences doubles entraîne une distribution différente des cycles de longueur minimum. Dans le cas des codes simplifiés, cette non-uniformité de la distribution des cycles sur les liens semble causer des baisses de performance sur certaines connexions au registre à décalage.

L'ensemble de ces observations se confirment pour les codes avec $J = \{8, \ldots, 11\}$

6.5 Étude du CMM

A l'origine, ce mémoire devait être consacré en partie à l'étude des cycles des codes CSO2C. La raison principale était de trouver une explication aux comportements des codes présentés dans le tableau 6.3. Cette partie est consacrée à expliquer ce problème.

Id	J	δ	Connexions $\mathcal{A} = \{\alpha_0, \alpha_1, \ldots \alpha_{J-1}\}$							6-cyc	8-cyc	ge	Perf. d'erreur
1	7	0.42	0	1	7	50	59	78	82	222	848	6,258	-
2	7	0.43	0	2	23	45	72	79	82	137	444	6,653	+
3	7	0.44	0	4	23	32	75	76	82	170	485	7,022	++

Tableau 6.3: Trois codes simplifiés $J = 7$ de même caractéristiques

Ce tableau reprend trois codes simplifiés, avec un coefficient de simplification quasi identique, une matrice génératrice de même taille et de même distance minimale $J + 1$.
À haut SNR, les performances de ces trois codes sont classées par ge. Les autres méthodes de calcul de CMM, basées sur les nœuds de variable ou parité donnaient des valeurs qui tendent vers le cycle minimum et qui ne pouvaient pas être utilisées pour ces codes. En règle générale, à cause du nombre élevé de cycles dans les codes convolutionnels, seul le calcul du ge donne des valeurs exploitables.
Les courbes de performances d'erreur obtenus par simulations de ces trois codes sont données Figure 6.11. Le simulateur utilisé est celui de Roy [4].

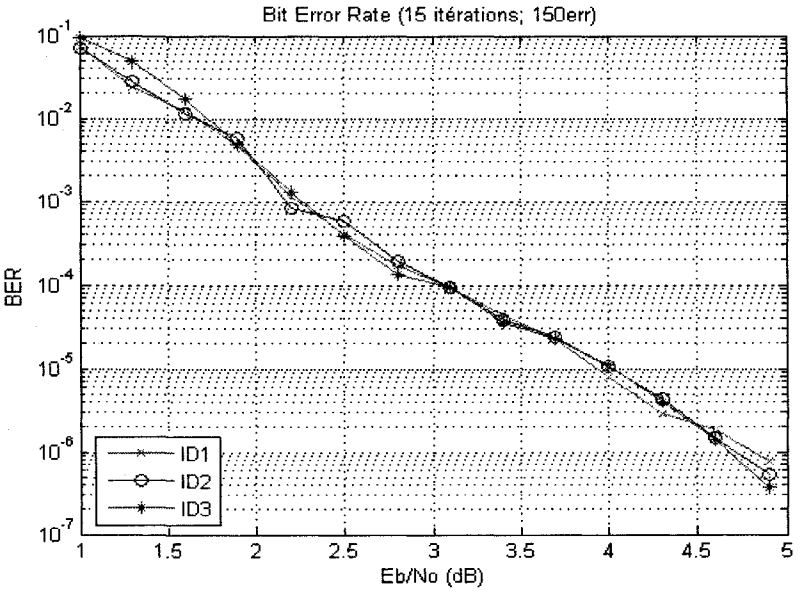


Figure 6.11: Performances d'erreur des codes S-CSO2C-WS du tableau 6.3 à la 15^{éme} itération de l'algorithme de décodage

Il est intéressant de voir que comme pour les codes LDPC, le calcul du CMM pourrait permettre d'obtenir une légère idée des performances sans simulation. Cependant, ce résultat est à prendre avec des pincettes puisque les performances d'erreur des trois codes sont très similaires et que cette étude n'a pas été réalisée sur d'autres codes (Il aurait fallu refaire tourner les algorithmes de recherche exhaustive de [4]).

CHAPITRE 7
CONCLUSION

7.1 Bilan de la recherche réalisée

Ce mémoire, qui est un travail nouveau, a permis de mieux cerner l'importance des cycles pour le décodage des codes en graphes.

Tout d'abord, nous avons expliqué en détail le fonctionnement de l'algorithme de décodage Somme-Produit pour que le lecteur comprenne pourquoi les petits cycles dégradent les performances.

Par la suite, nous avons étudié les nombres de cycles et développés des méthodes d'analyse adaptée pour l'étude des codes LDPC et CSO2C:

- Implémentation de notre propre méthode pour le comptage des cycles minimaux.
- Implémentation d'un algorithme surement existant d'exploration de graphe pour le comptage des cycles jusqu'à la longueur huit.
- Implémentation d'une méthode de comptage des cycles minimums moyens et mise en relation avec les performances d'erreur.
- Élaboration et test de notre méthode d'optimisation de matrice de parité (PER)
- Création d'un modèle mathématique pour l'évaluation du nombre de cycles des codes CSO2C-WS et LDPC.
- Mise en évidence d'irrégularités cycliques dans les codes doublement orthogonaux simplifiés.

L'ensemble de ces résultats permet d'établir définitivement l'existence d'un lien fort entre les cycles et les performances d'erreur. Ainsi, les méthodes d'analyse développées pourraient être utilisées pour la recherche de codes performants comme cela fut le cas avec le développement de l'algorithme PER.

7.2 Améliorations envisageables

Dans un premier temps, la modification de l'algorithme d'optimisation de matrice de parité (PER) pourrait être fructueuse. L'idée serait de déplacer les liens dégradant les performances au lieu de les supprimer. On rappelle que nous avons mis en évidence que pour des liens engagés dans des cycles minimaux à 6, il apparaît que la présence du lien apporte plus d'amélioration que de dégradation dans les performances d'erreur. Cela permettrait de créer des codes avec un cycle minimum à huit et d'en observer les performances d'erreur.

Deuxièmement, la généralisation de l'algorithme de comptage pour des cycles de longueur $g+2$ afin d'obtenir plus d'informations sur les cycles. Nous avons eu l'idée d'une méthode de comptage qui s'est avérée particulièrement efficace et très peu gourmande en mémoire. Elle permet de trouver le nombre exact de cycles minimaux sans avoir à passer à travers tous les cycles grâce à la formule (4.6). Pour comptabiliser les cycles à la longueur $g+2$, il est seulement nécessaire de codifier les messages transmis sur les liens lorsque les cycles de longueurs g sont trouvés.

Enfin, une généralisation des fonctions d'analyses et de comptages pour des cycles de longueur $g+2$ pourraient être intéressante. Grâce à la modification de l'algorithme de comptage que nous venons de suggérer, il deviendrait possible de collecter plus d'informations sur le comportement des cycles des LDPC et CSO2C. Nous pensons en

particulier au comptage des cycles sur les liens du graphe de Tanner qui s'est avéré particulièrement intriguent lors de l'analyse comparative des codes CSO2C-WS et S-CSO2C-WS.

7.3 Ouverture

Les résultats présentés ont pour seule vocation d'enrichir notre compréhension des cycles lors de la phase de décodage.

Cependant, il pourrait être fructueux, dans le cadre de recherches futures, d'utiliser une autre approche en complément de l'étude des cycles. Lors de la simulation d'un code LDPC, il serait intéressant de mémoriser la provenance des erreurs sur le graphe de Tanner. Il serait alors possible de connaître les nœuds du graphe qui sont les plus souvent responsables des erreurs.

Ensuite, une étude des cycles sur les nœuds incriminés pourrait être menée afin de déterminer quels sont les liens qui causent ces erreurs et agir en conséquence en modifiant la matrice de parité. Cette méthode, bien que très lourde en calcul, pourrait permettre d'améliorer significativement les performances de codes existants.

BIBLIOGRAPHIE

[1] **Shannon, C. E.** "*A Mathematical Theory of Communication*", Bell Syst. Tech. J., vol. 27 pp. 379-423, 1948.

[2] **Gallager, R. G.** *"Low Density Parity Check Codes"*, M.I.T. Press, Cambridge, Massachusetts, 1963.

[3] **Neal, Radford M.** *Software for Low Density Parity Check Codes* . Dept. of Statistics and Dept. of Computer Science, University of Toronto , Aout 2006.

[4] **Roy, E.** *"Recherche et analyse de codes convolutionnels doublement orthogonaux simplifiés au sens large"*, Mémoire de maîtrise de l'Ecole Polytechnique de Montréal, 2006.

[5] **Cardinal, C.** *"Décodage à seuil itératif sans entrelacement des codes convolutionnels doublement orthogonaux"*, Thèse de Doctorat de l'Ecole Polytechnique de Montréal, 2001.

[6] **Proakis, J.** *Digital Communications, Third Edition*, McGraw-Hill, New-York, 1995.

[7] **Cardinal, C., Haccoun, D., Gagnon, F. et Batani, N.** *"Turbo Decoding Using Convolutional Self Doubly Orthogonal Codes"*, Proceedings of ICC99, pp. 113-117.

[8] **Berrou, C., Glavieux, A. et Thitimajshima P.** "*Near Shannon Limit Error Correcting Coding and Decoding: Turbo Codes*", Proceedings of ICC'93, pp.1064-1070, Mai 1993.

[9] **Dru, F.** *"Décodage à seuil itératif des codes convolutionnels doublement orthogonaux perforés et application aux modulations multiniveaux"*, Mémoire de maîtrise de l'Ecole Polytechnique de Montréal, 2001.

[10] **Bachelier, B.** *"Analyse et determination de codes doublement orthogonaux pour décodage à seuil itératif"*, Mémoire de maîtrise de l'Ecole Polytechnique de Montréal, 2000.

[11] **Hamming, R.** *"Error-detecting and error-correcting codes",* Bell Syst. Tech. J. 29 pp 147 à 160, 1950 .

[12] **Forney, G. D.** *"Introduction to Binary Block Codes",* MIT OpenCoursWare, Chap6, 2005.

[13] **Lin, S., Costello, D. J. Jr.** *Error Control Coding: Fundamentals and Applications,* Prentice-Hall, Englewood Cliffs, N.J., 1983.

[14] **Tanner, R. M.** *"A recursive approach to low complexity codes",* IEEE Transactions on Information Theory 27:533–547, 1981.

[15] **Pearl, J.** *"Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference".* San Francisco, CA : Morgan Kaufmann, 1988.

[16] **Richardson, T. and Urbanke, R.** *"Modern Coding Theory",* Cambridge University Press, 2007.

[17] **Forney, G. D.** *"Codes on Graphs: Normal Realisation",* IEEE Trans. Inform. Theory, Feb. 2001.

[18] **Luby M., Mitzenmacher M., Shokrollahi A. and Spielman D.** *"Analysis of low-density codes and improved designs using irregular graphs",* Proceedings of the 30th ACM Symposium on Theory of Computing, 1998. pp. 249-258.

[19] **MacKay, D.J.C and Neal, R.M.** *"Good Codes Based on Very Sparse Matrices".* Berlin, Germany: Springer : 5th IMA Conference on Cryprography and Coding, 1995.

[20] **Luby M. G., Mitzenmacher M., Amin Shokrollahi M., and Spielman D. A.** *"Improved Low-Density Parity-Check Codes Using Irregular Graphs",* IEEE Trans. Inform. Theory, VOL. 47, NO. 2, Feb. 2001.

[21] **Hu X., Eleftheriou E., Arnold D.M.** *"Regular and irregular progressive edgegrowth Tanner graphs",* IEEE Trans. on Inform. Theory, 51(1):386–398, January 2005.

[22] **Brack T., Alles M., Kienle F., Wehn N.** *A synthesizable IP core for WIMAX 802.16E LDPC code decoding,* University of Kaiserslautern, IEEE 2006.

[23] **Johnson, S. and Weller S.** *"A family of irregular LDPC codes with low encoding complexity",* University of Newcastle, Callaghan, AUSTRALIE, IEEE communications letters, vol. 7, no2, pp. 79-81, 2003.

[24] **Jun F., Yang X.** *"A method of counting the number of cycles in LDPC codes",* ICSP2006 Proceedings, 2006.

[25] **Shokrollahi, A.** *"LDPC Codes: An Introduction",* Digital Fountain Inc, 2003.

[26] **Futaki, H., ohtsuki, T.** *"Performance of Low Density Parity Check (LDPC) Coded OFDM systems* . Tokyo : VTC 2001.

[27] **Halford, T. R. et Chugg, K.M.** *"An Algorithm for Couting Short Cycles in Bipartite Graphs",* IEEE Transactions on Information Theory VOL. 52, NO.1, P287, Jan. 2006.

[28] **Jun F., Yang X.** *"A method of counting the number of cycles in LDPC codes",* ICSP2006 Proceedings, 2006.

[29] **Hu, Xiao-Yu.** *Progressive Edge Growth construction,* IBM Zurich research labs, 2005.

[30] **Zhang, H. and Moura, J.M.F.** *Large-Girth LDPC codes based on graphical models,* Carnegie Mellon University, Pittsburg, 2005.

[31] **MacKay, D. J.C.** *Encyclopedia of Sparse Graph Codes,* Cavendish Laboratory, University of Cambridge, http://www.inference.phy.cam.ac.uk/mackay/codes/, 2005

[32] **Chung S.Y., Forney G. D., Richardson T. J., Urbanke R.** *"On the design of low density parity check codes within 0.045 dB of the Shannon limit",* IEEE Communication Letter, vol. 5, pp. 58-60, Feb 2001.

[33] **Neal, R.M.** *"LDPC Softwares",* University of Torronto, http://www.cs.utoronto.ca/~radford, 2005.

[34] **Haccoun, D., Cardinal, C., Gagnon, F.** *"Search and Determination of Convolutional Self-Doubly Orthogonal Codes for Iterative Threshold Decoding".* IEEE Transactions on Communications, 53(5), p. 802-809, 2005.

ANNEXE I :

CONSTRUCTION DU CODE LDPC DE GALLAGER

Dans sa thèse [2], Gallager a construit et utilisé le code régulier (20, 3, 4) pour illustrer sa méthode de construction de matrice de parité. L'idée était d'utiliser de petites matrices contenant un seul 1 par ligne pour générer une matrice plus grande. Ceci revient à diviser la matrice de parité en sous matrices.

Les codes réguliers sont définis par le triplet (n, d_n, d_r) avec le taux de codage suivant :

$$R = 1 - \frac{d_n}{d_r} \tag{I.1}$$

avec d_n, le degré des nœuds de variable et d_r, le degré des nœuds de parité.
On obtient alors la matrice Figure I.1 :

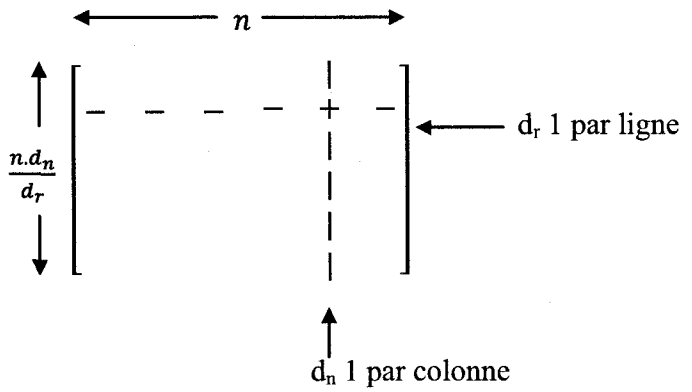


Figure I.1: Matrice de parité de Gallager

La matrice de parité **H** est ensuite divisée en d_r, d_n sous matrice de dimension $\frac{n}{d_r}, \frac{n}{d_r}$

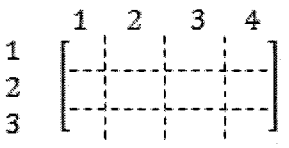


Figure I.2: Division de la matrice principale en sous matrice

L'algorithme de construction se déroule ensuite de la manière suivante :

→ **Étape 1 :**

- On place des matrices identités dans les sous matrices de la ligne 1 et de la colonne 1.
- On place la matrice suivante dans les autres sous matrices.

$$\begin{bmatrix} N & A & A & A & A \\ A & N & A & A & A \\ A & A & N & A & A \\ A & A & A & N & A \\ A & A & A & A & N \end{bmatrix}$$

où **N** = position Non acceptable

et **A** = position Acceptable.

Les positions non acceptables sont celles qui entraînent la création de cycles de longueur 4 avec les 1's déjà placés. La matrice suivante est alors obtenue :

$$H=\left[\begin{array}{ccccc:ccccc:ccccc:ccccc}
1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0\\
0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0\\
0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0\\
0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0\\
0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1\\
\hdashline
1&0&0&0&0&N&A&A&A&A&N&A&A&A&A&N&A&A&A&A\\
0&1&0&0&0&A&N&A&A&A&A&N&A&A&A&A&N&A&A&A\\
0&0&1&0&0&A&A&N&A&A&A&A&N&A&A&A&A&N&A&A\\
0&0&0&1&0&A&A&A&N&A&A&A&A&N&A&A&A&A&N&A\\
0&0&0&0&1&A&A&A&A&N&A&A&A&A&N&A&A&A&A&N\\
\hdashline
1&0&0&0&0&N&A&A&A&A&N&A&A&A&A&N&A&A&A&A\\
0&1&0&0&0&A&N&A&A&A&A&N&A&A&A&A&N&A&A&A\\
0&0&1&0&0&A&A&N&A&A&A&A&N&A&A&A&A&N&A&A\\
0&0&0&1&0&A&A&A&N&A&A&A&A&N&A&A&A&A&N&A\\
0&0&0&0&1&A&A&A&A&N&A&A&A&A&N&A&A&A&A&N
\end{array}\right]$$

Figure I.3: Étape 1 de la construction de la matrice (20, 3, 4) de Gallager

→ **Étape 2 :**

On prend une par une les sous-matrices restantes et on y effectue les tâches suivantes :

- Sur la première ligne, on place un 1 sur une position acceptable et on met des 0 sur toutes les autres positions de la ligne et de la colonne choisie.
- Pour éviter la formation de cycle de longueur 4, on change toutes les positions susceptibles de former un cycle avec le 1 ajouté, en position non acceptable.
- On refait les deux étapes précédentes pour les autres lignes de la sous-matrice avec un seul 1 par ligne et colonne.

$H=$

1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0
0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0
0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0
0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1
1	0	0	0	0	**0N**	**1**	**0A**	**0A**	**0A**	N	**N**	A	A	A	N	**N**	A	A	A
0	1	0	0	0	A	**0N**	A	A	A	A	N	A	A	A	A	N	A	A	A
0	0	1	0	0	A	**0A**	N	A	A	A	A	N	A	A	A	A	N	A	A
0	0	0	1	0	A	**0A**	A	N	A	A	A	A	N	A	A	A	A	N	A
0	0	0	0	1	A	**0A**	A	A	N	A	A	A	A	N	A	A	A	A	N
1	0	0	0	0	N	A	A	A	A	N	A	A	A	A	N	A	A	A	A
0	1	0	0	0	A	N	A	A	A	A	N	A	A	A	A	N	A	A	A
0	0	1	0	0	A	A	N	A	A	A	A	N	A	A	A	A	N	A	A
0	0	0	1	0	A	A	A	N	A	A	A	A	N	A	A	A	A	N	A
0	0	0	0	1	A	A	A	A	N	A	A	A	A	N	A	A	A	A	N

Figure I.4: Début de l'étape 2 de la construction de la matrice (20, 3, 4) de Gallager

Pour bien comprendre le fonctionnement de l'algorithme, la Figure I.4 illustre l'ajout du premier 1 dans l'étape 2, pour la première des 6 sous-matrices. On remarque que deux positions des autres sous-matrices sont affectées par cet ajout. Leur statut passe de position acceptable à position non acceptable pour empêcher la formation de cycles de longueur 4.

Enfin, la Figure I.5 illustre la matrice **H** une fois que l'algorithme est bloqué en appliquant l'étape 2. On remarque qu'il est impossible de continuer plus loin. L'étape 3 est alors utilisée pour débloquer la situation et permettre à l'algorithme de continuer d'appliquer l'étape 2.

$$H=\left[\begin{array}{ccccc:ccccc:ccccc:ccccc}
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
\hdashline
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\hdashline
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0N & 0A & 0N & 0N & 1 & N & A & N & N & N \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0N & 0A & 0N & 0N & N & N & A & N & N \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0N & 0A & 0N & 1 & 0N & N & N & N & N & N \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0N & 0N & 1 & 0N & 0A & N & N & N & N & A \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0A & N & 0N & 0A & 0N & A & N & N & N & N
\end{array}\right]$$

Figure I.5: Blocage à l'étape 2 de la construction de la matrice (20, 3, 4) de Gallager

→ **Étape 3:**

Lorsqu'il n'y a plus de position acceptable sur la ligne de la sous-matrice où l'on doit placer un 1, il est nécessaire de modifier la méthode de construction pour achever la matrice de parité :

- On recherche une position sur la ligne en cours L_i, égale à: $H(L_i, C_i) = N$.
- Puis, on recherche une ligne $L_i < L_a$ tel que :

$$\begin{cases} H(L_j, C_i) = H(L_i, C_j) = '0A' \\ H(L_j, C_j) = 1 \end{cases}$$

- Ensuite, on remplace par les valeurs suivantes :

$$\begin{cases} H(L_j, C_i) = H(L_i, C_j) = 1 \\ H(L_j, C_j) = '0A' \end{cases}$$

- Enfin, comme dans la procédure normale, on change toutes les positions acceptables en positions non acceptables pour empêcher les cycles de longueur 4.

$H=$

1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0
0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0
0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0
0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1
1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0
0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1
0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0
0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	1	0	0	0
0	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0
1	0	0	0	0	0	0	1	0	0	0N	0A	0N	0N	1	N	A	N	N	N
0	1	0	0	0	0	0	0	1	0	1	0N	0A	0N	0N	N	N	A	N	N
0	0	1	0	0	0	0	0	0	1	0N	**1**	0N	**0A**	0N	N	N	N	**A**	N
0	0	0	1	0	1	0	0	0	0	0N	0N	**1**	0N	0A	N	N	N	N	A
0	0	0	0	1	0	1	0	0	0	0A	N	0N	**1**	0N	A	N	N	N	N

Figure I.6: Matrice de parité après le déblocage de l'étape 3

Après quelques efforts supplémentaires, la matrice de parité de Gallager (20, 3, 4) est obtenue.

$$H=\left[\begin{array}{ccccc:ccccc:ccccc:ccccc}
1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0\\
0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0\\
0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0\\
0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0\\
0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1\\
\hdashline
1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0\\
0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1\\
0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&1&0&0&0&0\\
0&0&0&1&0&0&0&0&0&1&1&0&0&0&0&0&1&0&0&0\\
0&0&0&0&1&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0\\
\hdashline
1&0&0&0&0&0&0&1&0&0&0&0&0&0&1&0&1&0&0&0\\
0&1&0&0&0&0&0&0&1&0&1&0&0&0&0&0&0&1&0&0\\
0&0&1&0&0&0&0&0&0&1&0&1&0&0&0&0&0&0&1&0\\
0&0&0&1&0&1&0&0&0&0&0&0&1&0&0&0&0&0&0&1\\
0&0&0&0&1&0&1&0&0&0&0&0&0&1&0&1&0&0&0&0
\end{array}\right]$$

Figure I.7: Matrice de parité de Gallager (20, 3, 4)

ANNEXE II

AMELIORATIONS DES PERFORMANCE D'ERREUR PAR LA SUPRESSION DE PETIT CYCLE: COURBES

Cette annexe présente les courbes de performance d'erreur des codes de la série D, générés avec l'algorithme de [3] puis optimisés avec la méthode décrite au chapitre 5. Les tableaux II.1 et II.2 résument les caractéristiques de ces matrices de parités.

Code ID	D4	D1	D3	D10	D7	D2	D9	D14	D12	D5	D11	D13	D8	D6
4-Cycle	32	23	31	27	16	24	25	23	18	22	22	30	28	27
Distrib3D 1'	118	78	106	92	62	92	96	78	70	88	86	112	98	90
Distrib3D 2'	5	7	9	8	1	2	2	7	1	0	1	4	7	9
gnv	5.16	5.39	5.23	5.34	5.52	5.33	5.28	5.42	5.50	5.38	5.40	5.29	5.27	5.30
ge	5.46	5.64	5.47	5.57	5.76	5.56	5.53	5.64	5.71	5.62	5.64	5.66	5.52	5.55
O	82	2	52	43	88	101	18	148	72	30	96	11	132	140
Performance	Bonne													Mauvaise

Tableau II.1: Codes (254, 128) généré avec l'algorithme de [3]

Code ID	D'1	D'2	D'3	D'4	D'5	D'6	D'7	D'8	D'9	D'10	D'11	D'12	D'13	D'14
6-Cycle	133	154	145	155	144	138	133	147	142	153	153	150	145	141
gnv	6.08	6.05	6.06	6.00	6.11	6.05	6.08	6.06	6.08	6.02	6.05	6.08	6.02	6.11
ge	6.26	6.16	6.17	6.11	6.22	6.22	6.23	6.22	6.23	6.14	6.18	6.18	6.16	6.25
Supprimés $opD_i > 1$	5	1	4	2	1	4	3	2	4	5	2	0	2	1
Supprimés $opD_i > 0$	27	15	23	23	17	26	24	22	27	23	21	22	21	21
Performance	Bonne													Mauvaise

Tableau II.2: Codes (254, 128) optimisés par l'algorithme PER présenté au chapitre 5

Les lignes de ces tableaux correspondent aux résultats des opérations effectuées par le programme d'analyse des cycles. Voici ci-dessous la traduction :

- *X-Cycle*: Nombre de cycles à la longueur minimale X.
- *Distrib3D Y'*: Nombre de liens hébergeant Y cycles.
- *gnv, ge*: Cycle minimum moyen du code.
- *O*: Numéro du générateur pseudo-aléatoire utilisé pour la création du code.
- *Supprimés Z*: Nombre de liens supprimés pour la création du code Z
- *Performance:* Classement approximatif des codes par BER, à 4.5dB .

L'ensemble des figures se composent de trois courbes distinctes :

- *Di*: Code généré par l'algorithme de [3].
- *opDi >1*: Code optimisé par l'algorithme présenté au chapitre 5. Seuls les liens utilisés dans au moins deux cycles sont supprimés. Le tableau II.2 rapporte le nombre de liens supprimés. Les performances d'erreur sont généralement plus mauvaise.
- *opDi >0*: Code optimisé dans lequel tout les 4-cycles sont supprimés. Les performances sont toujours meilleures que celles du code d'origine.

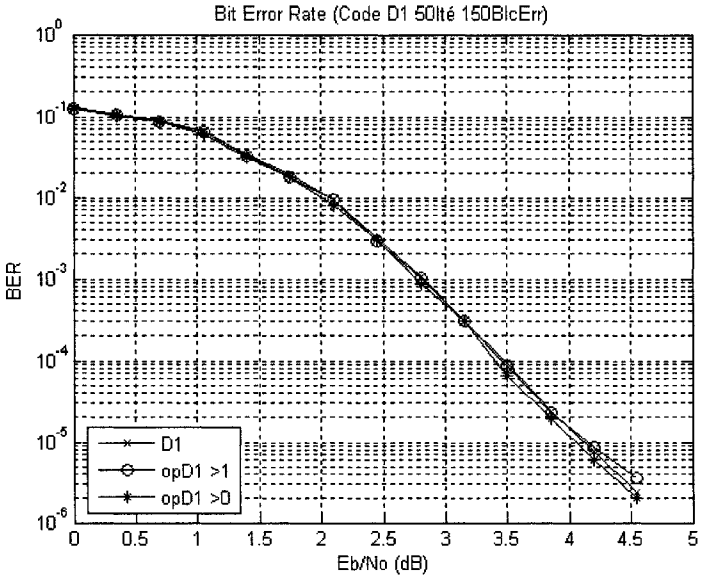


Figure II.1: Performances d'erreur du code D1

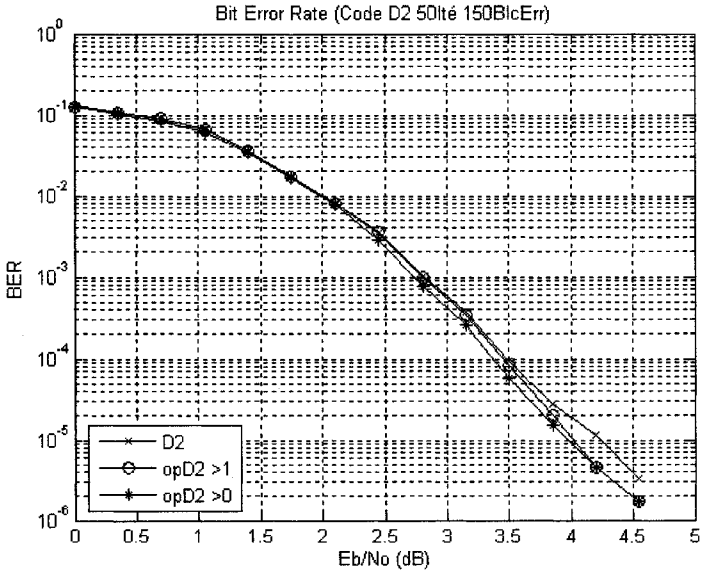


Figure II.2: Performances d'erreur du code D2

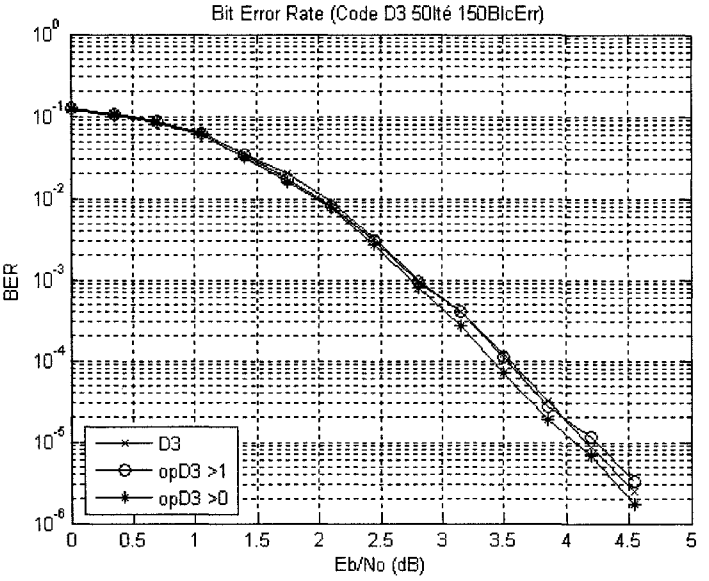


Figure II.3: Performances d'erreur du code D3

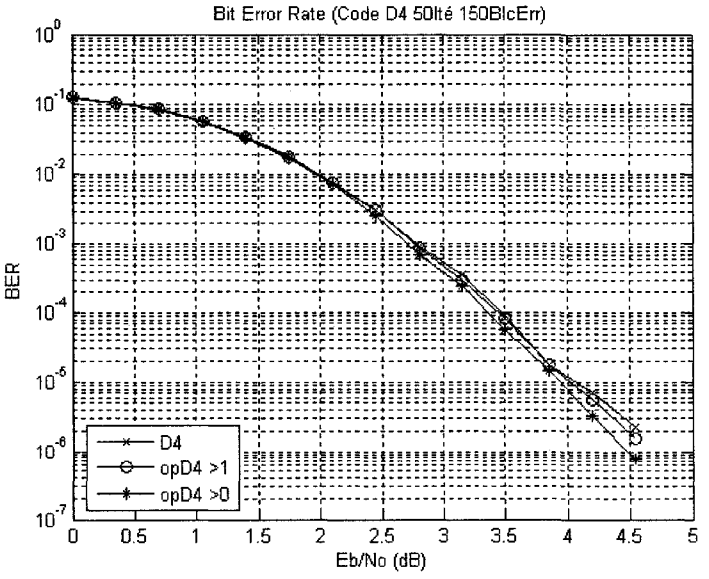


Figure II.4: Performances d'erreur du code D4

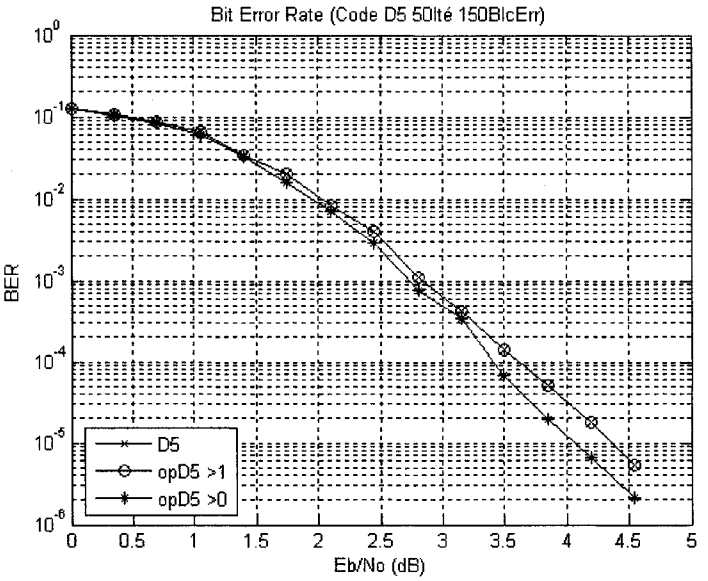


Figure II.5: Performances d'erreur du code D5

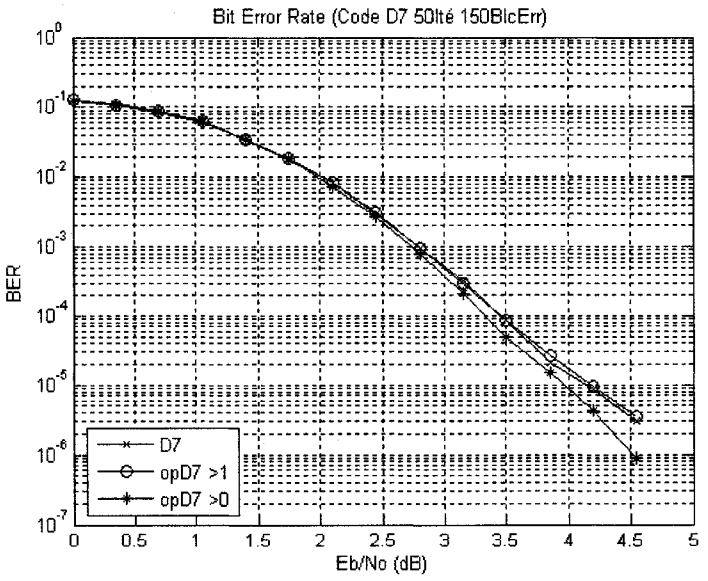


Figure II.6: Performances d'erreur du code D7

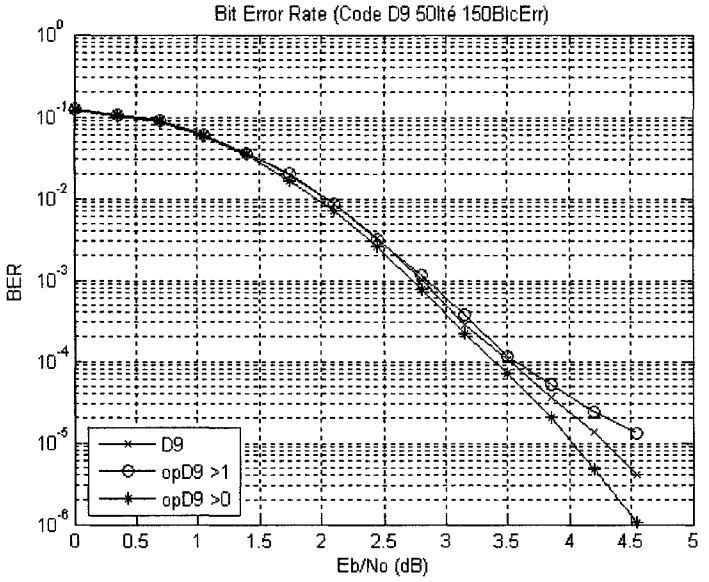


Figure II.7: Performances d'erreur du code D9

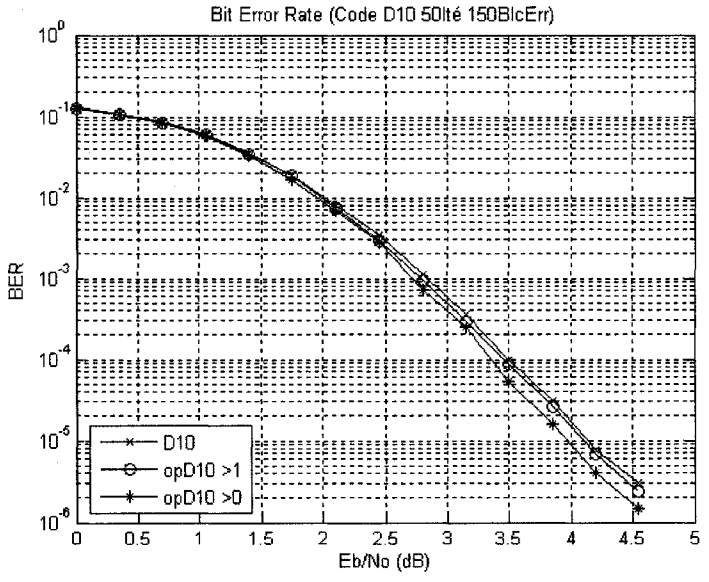


Figure II.8: Performances d'erreur du code D10

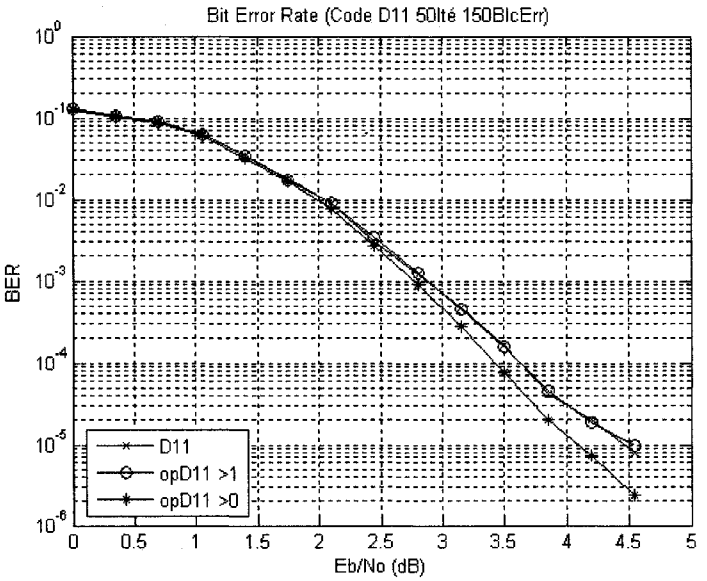


Figure II.9: Performances d'erreur du code D11

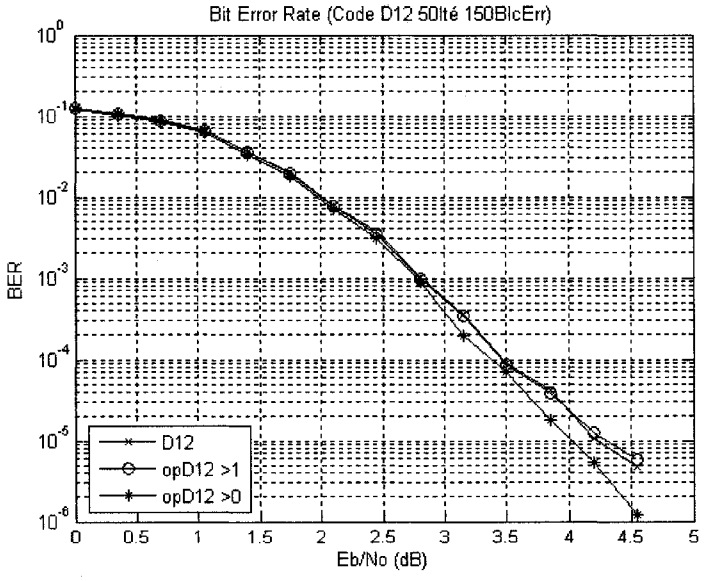


Figure II.10: Performances d'erreur du code D12

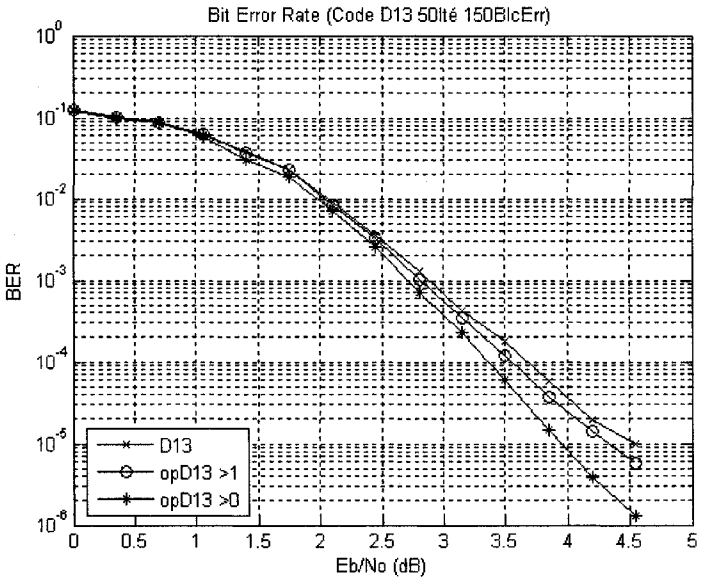


Figure II.11: Performances d'erreur du code D13

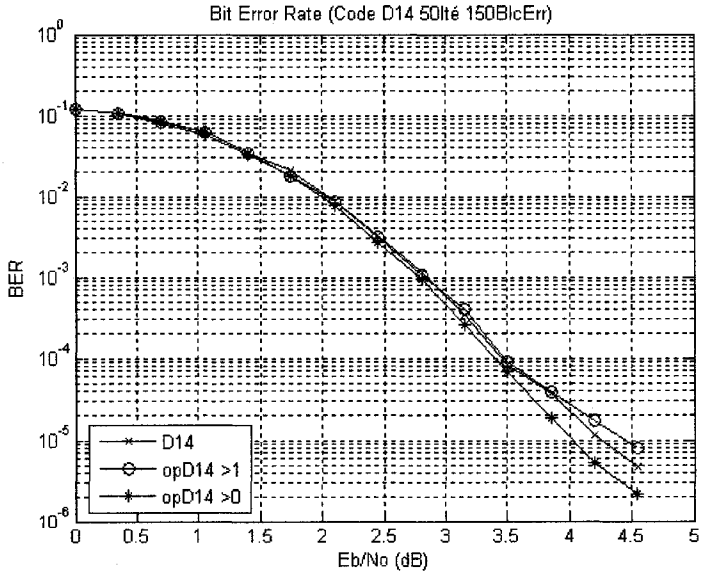


Figure II.12: Performances d'erreur du code D14

ANNEXE III

PERFORMANCES D'ERREUR ET NOMBRE DE CYCLES DES CODES CONVOLUTIONNELS DOUBLEMENT ORTHOGONAUX

Cette annexe présente les performances d'erreur et le nombre de cycles de longueur six et huit des codes convolutionnels utilisés dans ce mémoire [34][4]. L'ensemble $\mathcal{A} = \{\alpha_0, \alpha_1, \ldots \alpha_{J-1}\}$ représente les $J-1$ connexions entre le sommateur modulo-2 et les registres à décalages. Un exemple est donné Figure III.1.

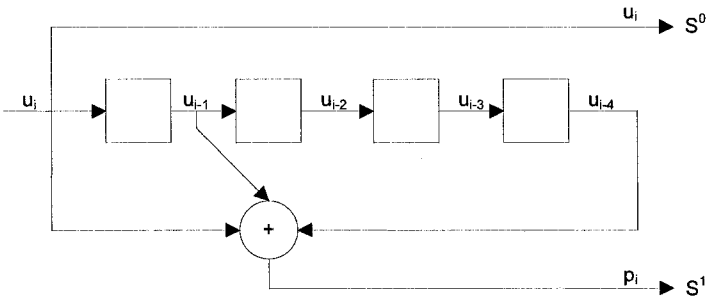


Figure III.1: Exemple de codeur convolutionnel systématique de taux de codage ½ où $\mathcal{A} = \{0, 1, 4\}$.

Le nombre de cycles de longueur six et huit ainsi que le CMM des liens sont donnés dans le tableau ci-après.

J	δ	Connexions $\mathcal{A} = \{\alpha_0, \alpha_1, \ldots \alpha_{J-1}\}$														6-cyc	8-cyc	ge
5	0	0	1	24	37	41	-	-	-	-	-	-	-	-	-	21	8	8,1477
5	0.2181	0	1	10	25	32	-	-	-	-	-	-	-	-	-	29	21	6,7846
5	0.3636	0	1	15	20	23	-	-	-	-	-	-	-	-	-	11	13	9,1529
5	0.3818	0	1	15	18	23	-	-	-	-	-	-	-	-	-	13	48	9,0345
6	0	0	1	17	70	95	100	-	-	-	-	-	-	-	-	147	136	6,283
6	0.2333	0	35	47	67	69	76	-	-	-	-	-	-	-	-	0	6	19,837
6	0.4333	0	2	11	26	42	45	-	-	-	-	-	-	-	-	73	274	6,7551
7	0	0	1	53	128	207	216	222	-	-	-	-	-	-	-	369	452	6,2759
7	0.2294	0	48	95	121	137	154	156	-	-	-	-	-	-	-	14	12	13,2917
7	0.3679	0	3	5	33	73	82	95	-	-	-	-	-	-	-	358	1599	6,369
7	0.4286	0	1	7	50	59	78	82	-	-	-	-	-	-	-	222	848	6,2067
7	0.4329	0	2	23	45	72	79	82	-	-	-	-	-	-	-	137	444	6,5429
7	0.4416	0	4	23	32	75	76	82	-	-	-	-	-	-	-	170	485	6,8065
8	0	0	43	139	322	422	430	441	459	-	-	-	-	-	-	426	190	7,4862
8	0.2241	0	40	157	200	249	253	275	287	-	-	-	-	-	-	148	121	8,7763
8	0.4261	0	4	19	44	95	130	144	147	-	-	-	-	-	-	587	2230	6,3792
8	0.4433	0	1	6	45	100	119	127	142	-	-	-	-	-	-	635	3324	6,1017
8	0.4828	0	9	22	55	95	124	127	129	-	-	-	-	-	-	343	1209	6,913
9	0	0	9	21	395	584	767	871	899	912	-	-	-	-	-	3827	9846	6,1182
9	0.2357	0	41	196	215	346	349	385	446	495	-	-	-	-	-	1582	9630	7,1725
9	0.3889	0	17	28	78	190	216	256	263	264	-	-	-	-	-	1024	3454	6,7444
9	0.4895	0	1	17	26	127	138	185	204	208	-	-	-	-	-	1506	10189	6,1014
10	0	0	29	40	43	1020	1328	1495	1606	1696	1698	-	-	-	-	14040	49950	6,1714
10	0.1363	0	128	261	410	534	698	743	891	1038	1190	-	-	-	-	11229	56991	7,0683
10	0.3575	0	2	10	31	103	219	316	370	447	454	-	-	-	-	6967	73847	6,0472
10	0.4106	0	95	113	145	291	346	400	424	443	453	-	-	-	-	1783	7840	8,3608
10	0.4638	0	1	5	12	61	140	251	294	327	352	-	-	-	-	6156	71116	6,0295
10	0.4917	0	1	87	93	226	262	296	316	327	340	-	-	-	-	1455	7462	6,2219
11	0	0	220	521	695	908	926	1059	2457	3367	3458	3490	-	-	-	*85796*	*492162*	6, 0153
11	0.0935	0	290	487	633	890	1153	1243	1542	1858	2117	2239	-	-	-	31183	193962	7,1802
11	0.1786	0	10	211	441	541	808	1067	1120	1256	1317	1582	-	-	-	20941	170193	6,1282
11	0.3071	0	117	130	174	192	443	643	717	801	808	923	-	-	-	14642	136084	7,0756
11	0.4539	0	1	5	12	32	61	199	350	434	480	588	-	-	-	22908	458025	6,0147
12	0	0	48	212	1014	1381	2217	4198	4373	4766	4885	4914	5173	-	-	66699	291992	6,0399
12	0.1922	0	6	79	280	482	693	972	1108	1483	1575	1998	2035	-	-	55043	*525691*	6,0399
12	0.2420	0	39	56	249	325	489	816	1183	1279	1368	1627	1631	-	-	37946	*319030*	6,1853
12	0.3311	0	14	16	60	124	319	481	708	906	1042	1048	1061	-	-	27585	*285570*	6,0987
12	0.4632	0	1	5	12	32	61	107	271	411	584	707	894	-	-	60746	*1794409*	6,0084
13	0.1418	0	480	818	826	1458	1562	1869	2519	2667	3260	3872	4154	4232	-	116556	*943538*	6,895
13	0.2798	0	228	242	297	523	586	785	796	1001	1635	1738	2052	2356	-	131111	*2237779*	6,6178
13	0.4193	0	2	12	144	190	207	633	747	974	1052	1111	1214	1217	-	35205	*401246*	6,0159
14	0.4269	0	1	5	12	32	61	107	230	355	514	919	1188	1660	1967	264514	*10890445*	6,0032

Tableau III.1: CMM des liens et nombre de cycles de longueur six et huit des codes CSO2C-WS et S-CSO2C-WS

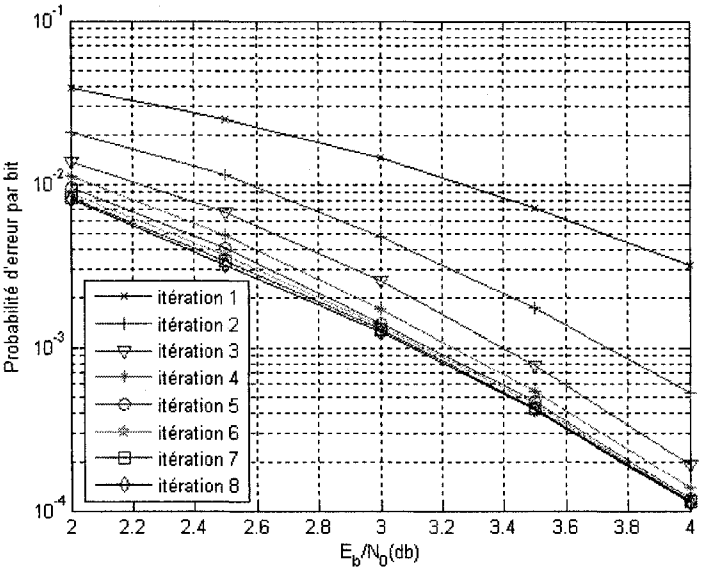


Figure III.2: Performance d'erreur du code $J = 6$ $A = \{0\ 1\ 15\ 20\ 23\}$

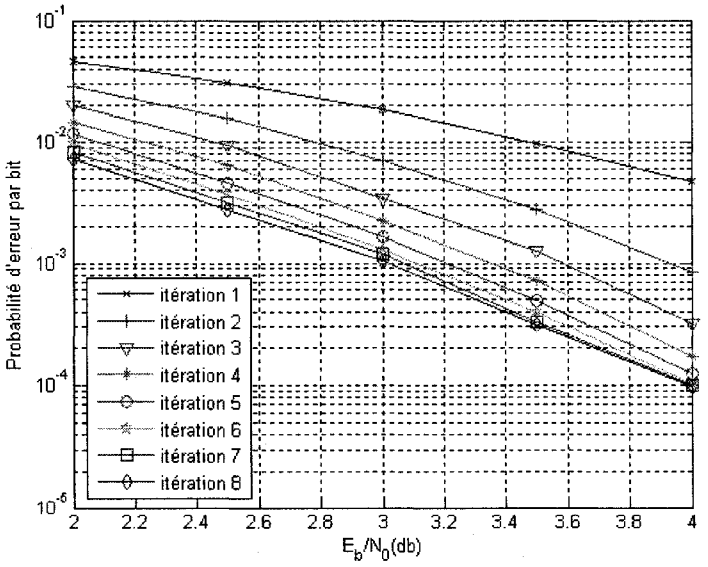


Figure III.3: Performance d'erreur du code $J = 5$ $A = \{0\ 1\ 24\ 37\ 41\}$

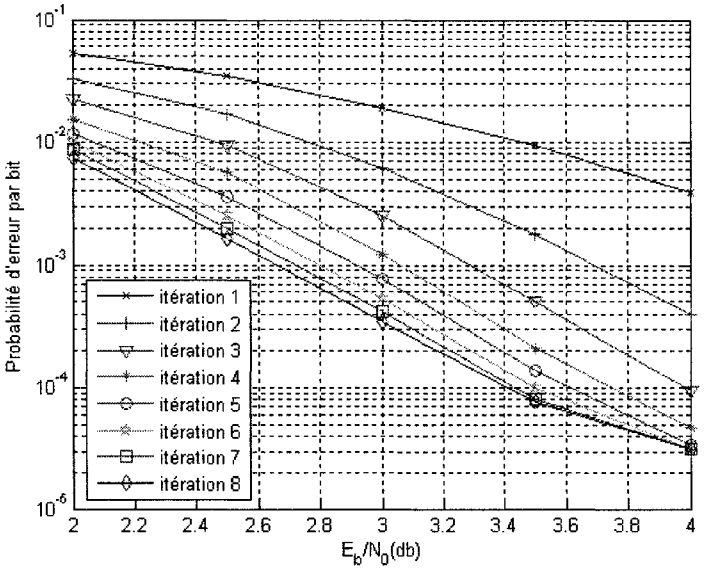


Figure III.4: Performance d'erreur du code $J = 6$ $A = \{0\ 2\ 11\ 26\ 42\ 45\}$

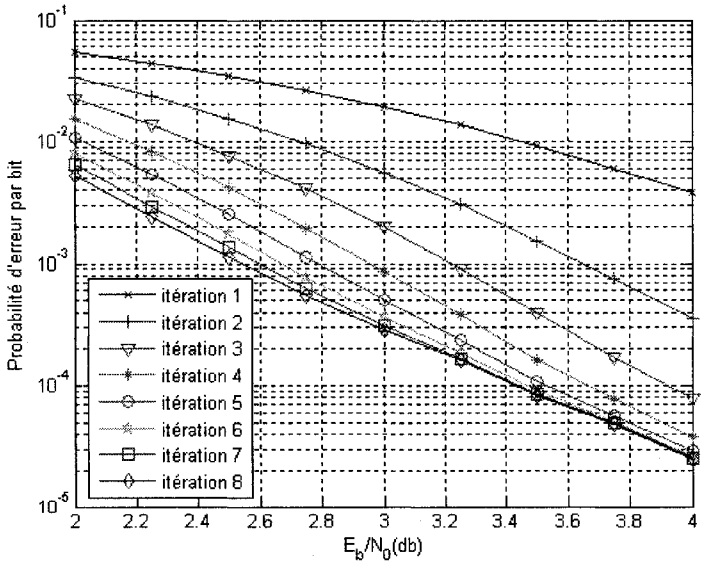


Figure III.5: Performance d'erreur du code $J = 6$ $A = \{0\ 35\ 37\ 67\ 69\ 76\}$

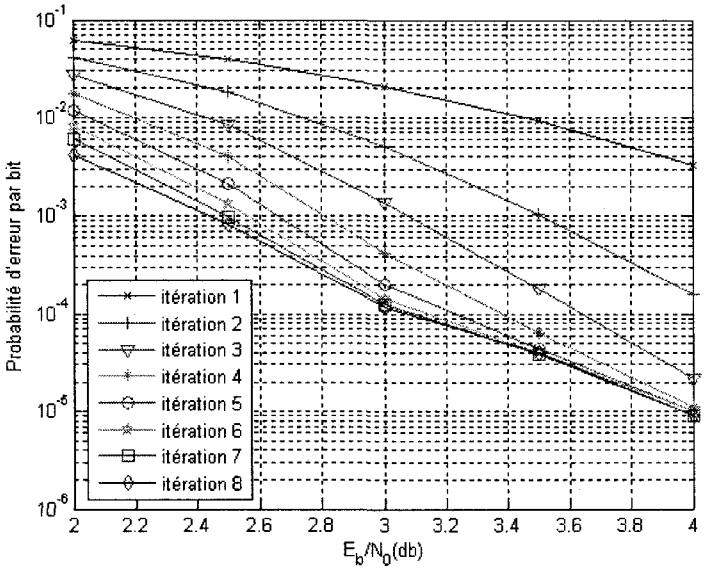


Figure III.6: Performance d'erreur du code $J = 7\ A = \{0\ 1\ 7\ 50\ 59\ 78\ 82\}$

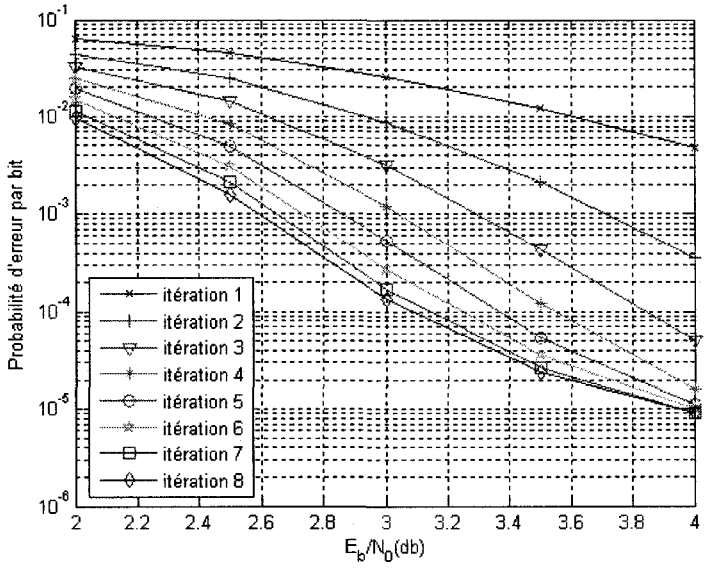


Figure III.7: Performance d'erreur du code $J = 7\ A = \{0\ 2\ 23\ 45\ 72\ 79\ 82\}$

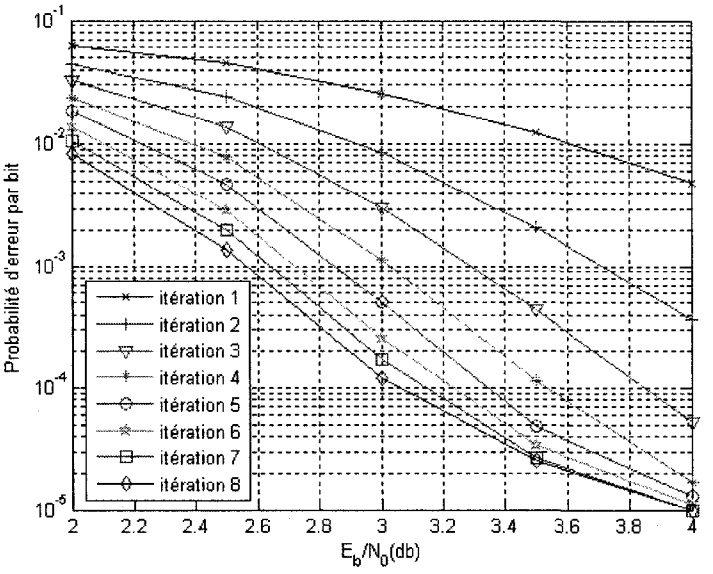


Figure III.8: Performance d'erreur du code $J = 7\ A = \{0\ 4\ 23\ 32\ 75\ 76\ 82\}$

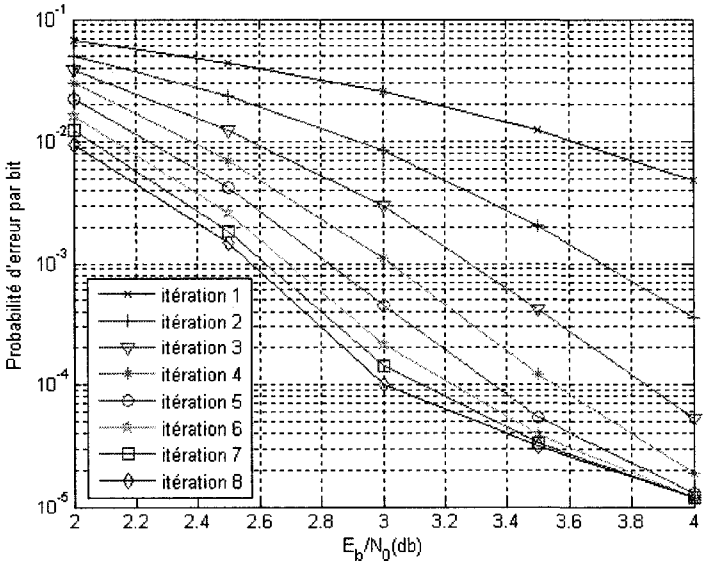


Figure III.9: Performance d'erreur du code $J = 7\ A = \{0\ 3\ 5\ 33\ 73\ 82\ 95\}$

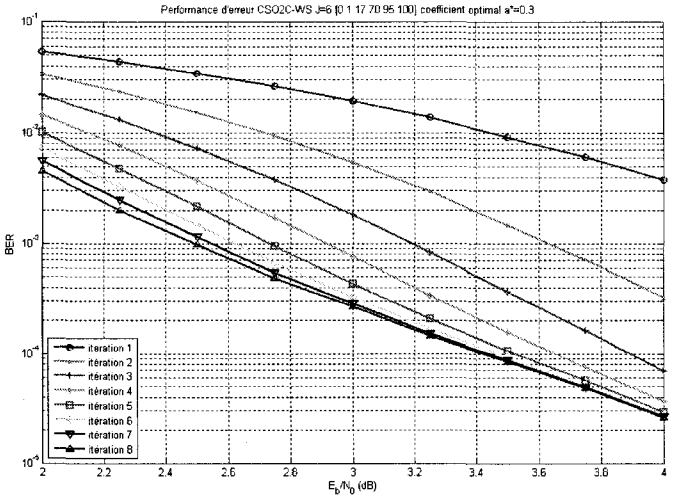


Figure III.10: Performance d'erreur du code $J = 6\ A = \{0\ 1\ 17\ 70\ 95\ 100\}$

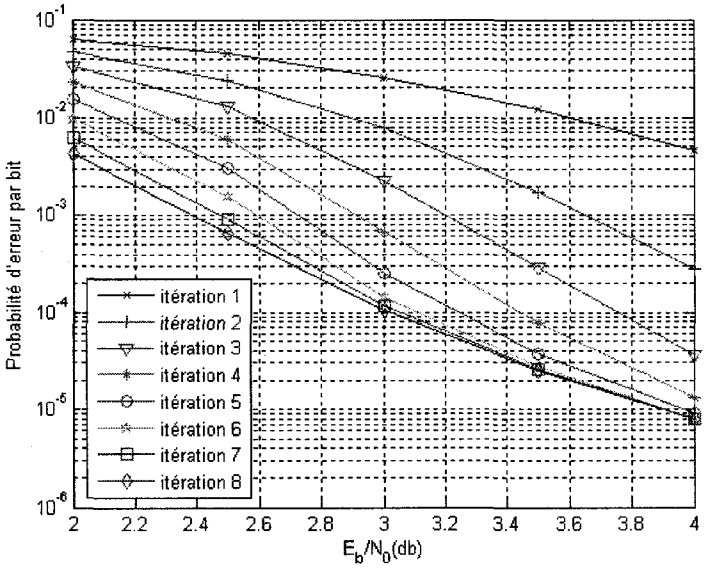


Figure III.11: Performance d'erreur du code $J = 7\ A = \{0\ 1\ 53\ 128\ 207\ 216\ 222\}$

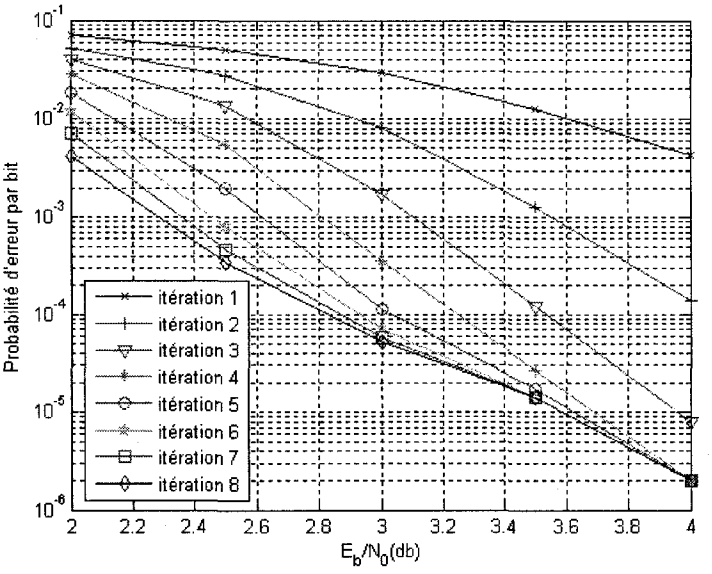


Figure III.12: Performance d'erreur du code $J = 8$

$A = \{0\ 40\ 157\ 200\ 249\ 253\ 275\ 287\}$

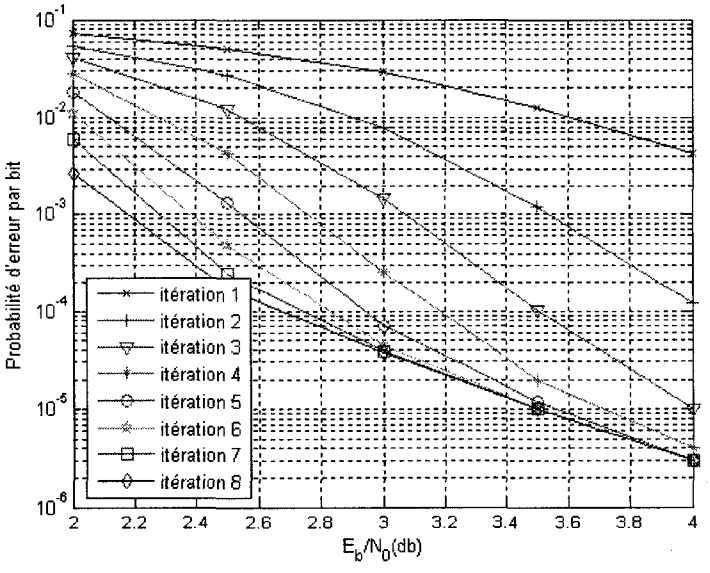


Figure III.13: Performance d'erreur du code $J = 8$

$A = \{0\ 43\ 139\ 322\ 422\ 430\ 441\ 459\}$

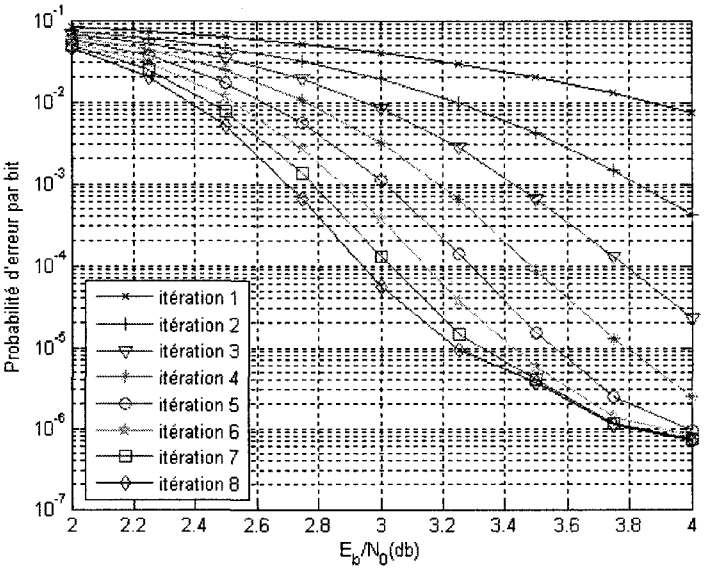


Figure III.14: Performance d'erreur du code $J = 9$

$A = \{0\ 1\ 17\ 26\ 127\ 138\ 185\ 204\ 208\}$

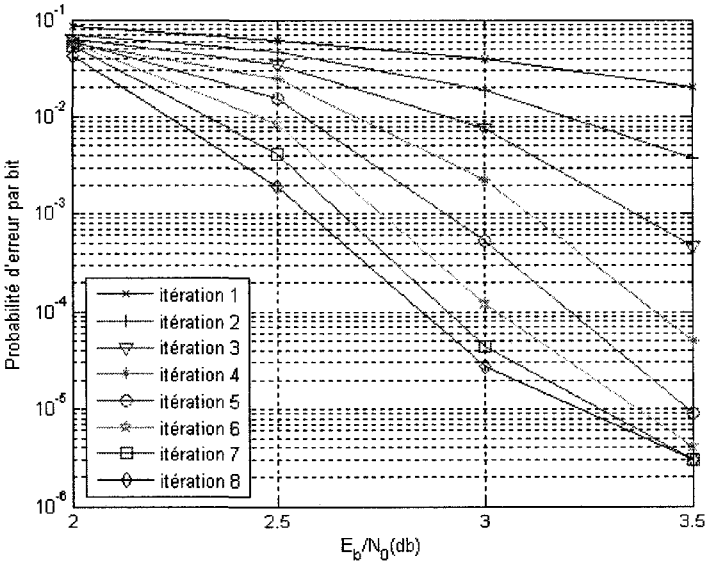


Figure III.15: Performance d'erreur du code $J = 9$

$A = \{0\ 41\ 196\ 215\ 346\ 349\ 385\ 446\ 495\}$

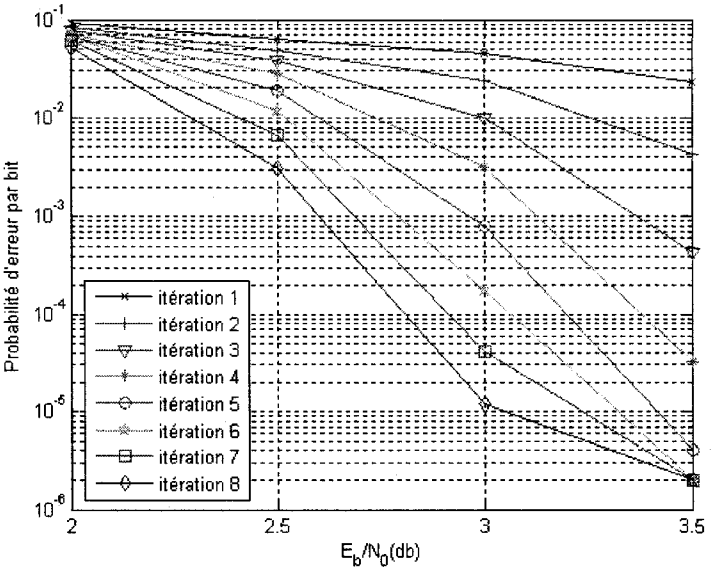


Figure III.16: Performance d'erreur du code $J = 10$

$A = \{0\ 1\ 5\ 12\ 61\ 140\ 251\ 294\ 327\ 352\}$

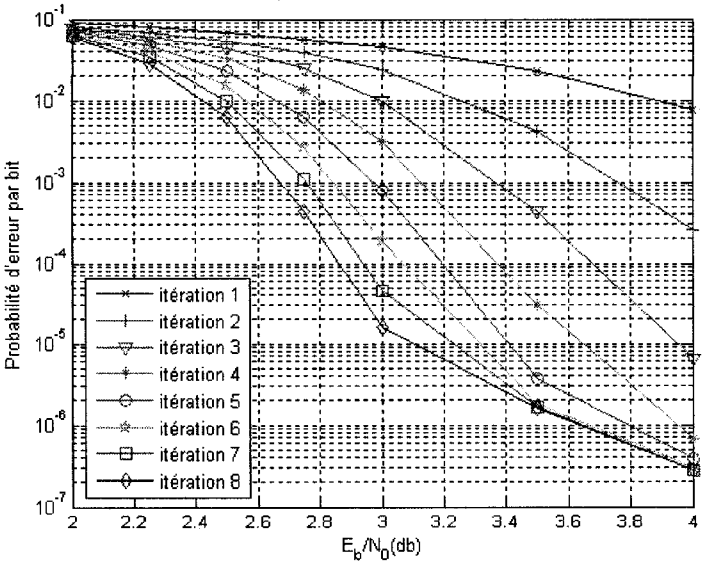


Figure III.17: Performance d'erreur du code $J = 10$

$A = \{0\ 1\ 87\ 93\ 226\ 262\ 296\ 316\ 327\ 340$

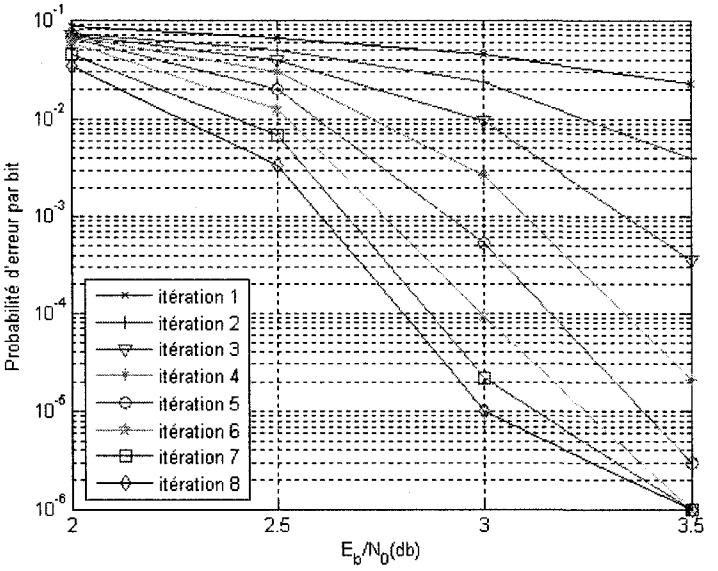


Figure III.18: Performance d'erreur du code $= 10$

$A = \{0\ 2\ 10\ 31\ 103\ 219\ 316\ 370\ 447\ 454\}$

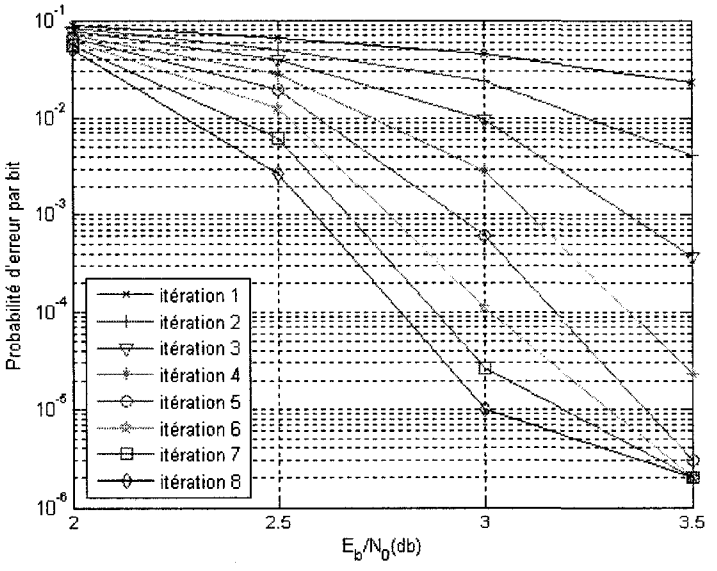


Figure III.19: Performance d'erreur du code $J = 10$

$A = \{0\ 95\ 113\ 145\ 291\ 346\ 400\ 424\ 443\ 453\}$

ANNEXE IV

BIBLIOTHEQUE DES CODES LDPC UTILISÉS

Cette annexe présente les caractéristiques de génération des matrices de parités pour les codes LDPC utilisés dans ce mémoire. Une copie identique des codes peut être retrouvée en suivant la procédure décrite dans cette annexe [33].

IV.1 Caractéristiques des codes utilisés

Voici les caractéristiques de génération des séries de code utilisés dans ce mémoire :

Code	A_{id}	B_{id}	C_{id}	D	T_{id}	Z_{id}
4-Cycle free	non	non	non	non	non	non
n	256	256	256	256	2200	2000
r	128	128	128	128	1100	1000
O	3	3	3	3	3	3
G	id	id	id	A II.1	id	id

Code	Ha	Hb	Hc	Hd	He	Xa_{id}	Xb_{id}	Xc_{id}	Ya_{id}	Yb_{id}	Yc_{id}
4-Cycle free	oui	non	non	non	non	non	non	non	non	non	non
n	1000	1000	1000	1000	1000	500	1000	2000	500	1000	2000
r	500	500	500	500	500	250	500	1000	250	500	1000
O	1	2	3	4	5	3	3	3	6	6	6
G	233 - 848					id	id	id	id	id	id

Tableau IV.1: Paramètres de construction des codes LDPC utilisé dans ce mémoire

avec :

- n, r: la taille de la matrice de parité.
- O: Le nombre de 1's par colonne.
- G: Numéro de la séquence pseudo-aléatoire utilisé pour la construction du code.

IV.2 Méthode pour générer une matrice de parité

Les matrices de parité contenant des 4-cycles peuvent être recrées à l'identique grâce aux séquences de nombre aléatoire pré-générées et stockées à la racine du programme utilisées pour leur création. Veuillez suivre la procédure suivante :

1. Démarrer une session Unix / Linux sur une machine, avec les droits d'administration.
2. Télécharger : http://www.cs.utoronto.ca/~radford/ftp/LDPC-2006-02-08/LDPC-2006-02-08.tar
3. Décompresser l'archive: `tar xf LDPC-2006-02-08.tar`
4. Installer l'archive:
```
cd LDPC-2006-02-08
make
```
5. L'installation est maintenant terminée. On peut alors générer un fichier binaire contenant la matrice de parité :
```
./make-ldpc xxx.pchk r n G evenboth O
./pchk-to-alist xxx.pchk xxx.alist
```

Bien évidement, remplacer 'xxx' par le nom du fichier, 'r', 'n' par la taille de la matrice de sortie, 'G' par le numéro de la séquence pseudo-aléatoire utilisée pour générer le code et enfin 'O' par le nombre de 1's par colonne.

Le fichier au format '*alist*' est compatible et importable par l'ensemble des programmes utilisés dans ce mémoire.